Animal_cv_neural_networks

March 31, 2024

Computer Vision Multi-Classification Neural Network

Ernest López Sánchez

We are going to apply a multi-classification neural network to a dataset of 26350 animal images in order to classify them by species.

There are 10 possible classes: - Butterfly - Cat - Chicken - Cow - Dog - Elephant - Horse - Sheep - Spider - Squirrel

First, we will explore the dataset.

Then we will transform our data as needed and compare the performance of different neural network architectures.

We are specifically going to design, tune, train and test:

- A fully connected traditional artificial neural network.
- A convolutional neural network with and without data augmentation.

We will also test the performance of InceptionV3 (through transfer learning), an already-trained convolutional neural network specifically designed for computer vision.

We will check how it does by itself, without any exposure to the actual data, and with expose to the actual data (fine-tuning).

Afterward, we are going to evaluate the results of our model using the typical metrics of a multi-classification problem.

Finally, we are going to try to improve the performance our first small convolutional neural network had by improving the neural network architecture.

We will use TensorFlow and the Keras functional API in order to do so. If you were interested in the specific coding behind any of the Keras layers, you can refer to the Keras documentation.

I'm using a paid GPU in Google Colab to train and test the models, but you can use the free ones given by TensorFlow-cuda.

Images are represented using RGB, so an image size will always be [height, width, 3].

[ ]:
```
import tensorflow as tf
print("TF version   : ", tf.__version__)
print("GPU available: ", tf.config.list_physical_devices('GPU'))
import keras
print("Keras version   : ", keras.__version__)
```

```
TF version   :  2.15.0
GPU available:  [PhysicalDevice(name='/physical_device:GPU:0',
device_type='GPU')]
Keras version   :  2.15.0
```

[ ]:
```python
# Import the desired Keras elements
from keras.utils import image_dataset_from_directory
from keras.layers import (
    GlobalAveragePooling2D, Flatten, Input,
    Dense, Dropout, Conv2D, Conv2DTranspose, BatchNormalization,
    MaxPooling2D, UpSampling2D, Rescaling, Resizing,
    RandomFlip, RandomRotation, RandomZoom, RandomContrast)
from keras.callbacks import (EarlyStopping, ReduceLROnPlateau)
from keras.optimizers import (Adam, RMSprop)
from keras import Sequential, Model
```

[ ]:
```python
# Import some utilities
import cv2
from PIL import Image
import glob
import os
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time
from sklearn.metrics import classification_report, confusion_matrix
import itertools
```

0.1 1. Download, analysis and preprocessing of the data

The dataset used can be downloaded at this Zenodo link: https://zenodo.org/records/10898524

It has the typical structure of a computer vision dataset: train and test directories, inside which there is a folder containing the images for each class.

Let's explore our dataset.

[ ]:
```python
# Load the file from google drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```python
# Check the classes in the train dataset
images_directory = '/content/drive/MyDrive/images'
train_classes = os.listdir(os.path.join(images_directory, 'train'))
print(f"The classes available in the train dataset are {sorted(train_classes)}")
```

```
The classes available in the train dataset are ['butterfly', 'cat', 'chicken',
'cow', 'dog', 'elephant', 'horse', 'sheep', 'spider', 'squirrel']
```

[ ]:
```python
# Check the classes in the test dataset
images_directory = '/content/drive/MyDrive/images'
test_classes = os.listdir(os.path.join(images_directory, 'test'))
print(f"The classes available in the test dataset are {sorted(test_classes)}")
```

```
The classes available in the test dataset are ['butterfly', 'cat', 'chicken',
'cow', 'dog', 'elephant', 'horse', 'sheep', 'spider', 'squirrel']
```

[ ]:
```python
# Check how many images per class the train dataset contains
def count_files(directory):
  ''' Given a directory counts the files that are inside'''
  return len([filename for filename in os.listdir(directory) if os.path.
 ↪isfile(os.path.join(directory, filename))])


images_directory = '/content/drive/MyDrive/images'
train_directory = os.path.join(images_directo
                               ry, 'train')
train_counts = {}
for class_name in sorted(os.listdir(train_directory)):
    class_path = os.path.join(train_directory, class_name)
    train_counts[class_name] = count_files(class_path)

print('The files for each class are:')
for class_name, count in train_counts.items():
    print(f"{class_name}: {count}")
```

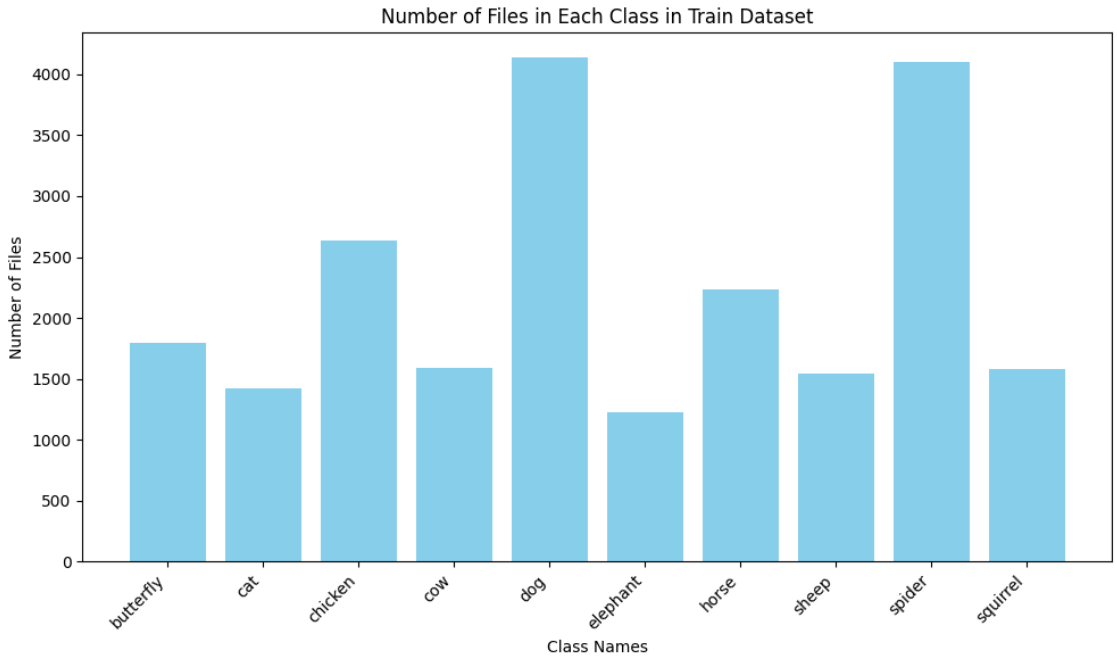
```
The files for each class are
butterfly: 1795
cat: 1418
chicken: 2633
cow: 1586
dog: 4134
elephant: 1229
horse: 2230
sheep: 1547
spider: 4098
squirrel: 1583
```

[ ]:
```python
# Check how many images per class the test dataset contains
test_directory = os.path.join(images_directory, 'test')

test_counts = {}
for class_name in sorted(os.listdir(test_directory)):
    class_path = os.path.join(test_directory, class_name)
```

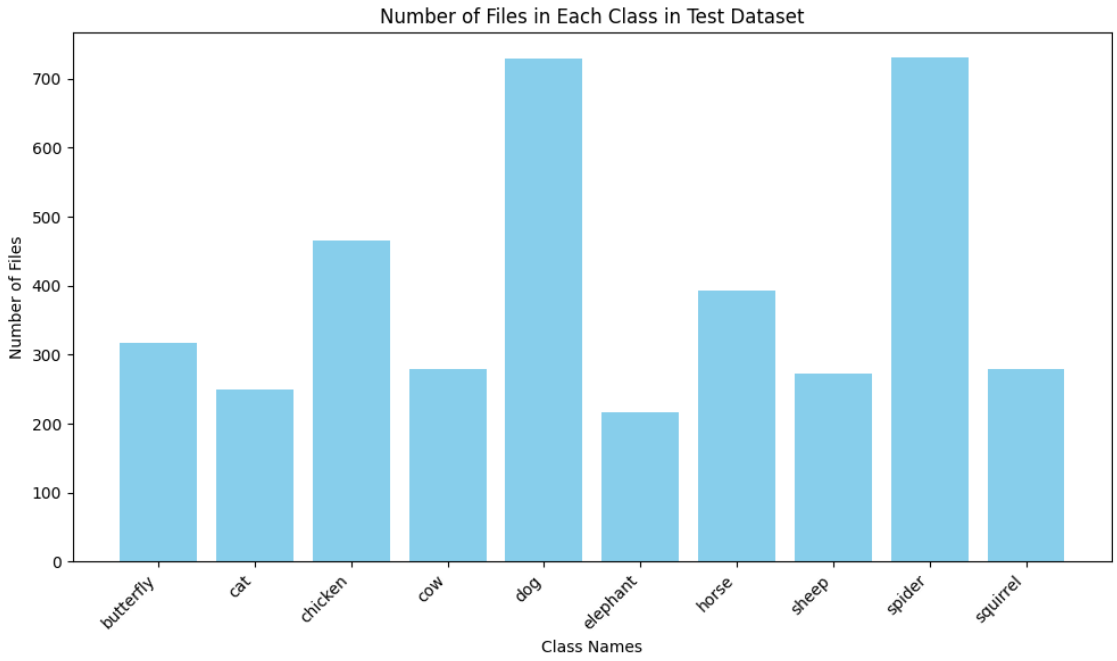
```
    test_counts[class_name] = count_files(class_path)

print('The files for each class are')
for class_name, count in test_counts.items():
    print(f"{class_name}: {count}")
```

```
The files for each class are
butterfly: 317
cat: 250
chicken: 465
cow: 280
dog: 729
elephant: 217
horse: 393
sheep: 273
spider: 730
squirrel: 279
```

[ ]:
```
# Plot the number of images for each class in the train dataset
class_names = sorted(list(train_counts.keys()))
counts = list(train_counts.values())

plt.figure(figsize=(10, 6))
plt.bar(class_names, counts, color='skyblue')
plt.xlabel('Class Names')
plt.ylabel('Number of Files')
plt.title('Number of Files in Each Class in Train Dataset')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



[ ]:
```
# Plot the number of images for each class in the test dataset
class_names = sorted(list(test_counts.keys()))
counts = list(test_counts.values())

plt.figure(figsize=(10, 6))
plt.bar(class_names, counts, color='skyblue')
plt.xlabel('Class Names')
plt.ylabel('Number of Files')
plt.title('Number of Files in Each Class in Test Dataset')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



The training dataset has 22.253 images, while the test dataset has 3.926. Thus, our training dataset has 85% of the data and the test has 15%.

The distribution of images for each class is exactly the same in both datasets. As an example, 8% of all images are of class butterfly in both datasets. We can visually check that the only thing that changes between both graphics is the scale of the y-axis.

This is done in order to prevent bias in training or testing. If this distribution of images per class was highly different in test than in training, we could find that our results are not optimal, or they can even be severely deficient because our model would have been trained with a data structure that is very different from the testing one.

Our training data should be able to provide the model with a generalization of reality; if the training data is far away from reality, then our model will not have good results.

By having the same distribution in both datasets, we make sure that our model does not underperform or overperform due to distribution issues in either of the two datasets. So we make sure he is not learning the distribution of samples per class; instead, he is learning the relationships and trends underlying our data.

0.1.1 1.2. Image visualization

Let's check some of our images.

[ ]:
```
# Randomly plot 10 images and its class.
import random

data_dir = '/content/drive/MyDrive/images/train'
```

```
class_dirs = os.listdir(data_dir)

plt.figure(figsize=(15, 10))
for i, class_dir in enumerate(class_dirs, start=1):
    class_path = os.path.join(data_dir, class_dir)
    images = os.listdir(class_path)

    random_image = random.choice(images)

    image_path = os.path.join(class_path, random_image)
    image = Image.open(image_path)

    plt.subplot(2, 5, i)
    plt.imshow(image)
    plt.title(f'Class: {class_dir}')
    plt.axis('off')

plt.show()
```

[ ]:
```
# Let's check the dynamic range of the images in our dataset.
import os
from PIL import Image
import random
data_dir = '/content/drive/MyDrive/images/train'
```

```
min_pixel_value = float('inf')
max_pixel_value = 0

num_images_to_sample = 10

# Get 10 random images
all_image_paths = []
for class_dir in os.listdir(data_dir):
    class_path = os.path.join(data_dir, class_dir)
    all_image_paths += [os.path.join(class_path, image_name) for image_name in␣
 ↪os.listdir(class_path)]
random.shuffle(all_image_paths)

# Check highest and lowest possible values of a pixel of those 10 images.
for image_path in all_image_paths[:num_images_to_sample]:
    image = Image.open(image_path)
    pixels = list(image.getdata())
    min_pixel_value = min(min_pixel_value, min(min(pixel) for pixel in pixels))
    max_pixel_value = max(max_pixel_value, max(max(pixel) for pixel in pixels))

print("Minimum pixel value:", min_pixel_value)
print("Maximum pixel value:", max_pixel_value)
```

```
Minimum pixel value: 0
Maximum pixel value: 255
```

If we check our image plotting, we can easily see that not all images have the same size. This is a problem; we need them to have a common size in order to feed them to our neural network.

So we need to resize them to have a common format.

The dynamic range of an image is the highest and lowest possible values of a pixel.

We check 10 images due to efficiency reasons; we assume it is enough to find the lowest and highest possible values of a pixel; if this was not correct, we would find it afterward.

The dynamic range of our images goes from 0 to 255; a common characteristic of RGB representations is that the range goes from (0,0,0) (black) to (255, 255, 255) (white).

More info on RGB: https://www.w3schools.com/colors/colors_rgb.asp

0.1.2 1.3. Creation of the datasets in TensorFlow/Keras format.

We are going to create our database in Keras/TensorFlow format using the images in our file.

The function we are going to use tf.keras.utils.image_dataset_from_directory() specifically expects the data to be stored in directories following the same structure as our file has.

As explained previously, it is a common structure for computer vision problems, and there are many functions that can load the data from that structure.

If you were interested in the documentation: https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_

We are going to create 3 datasets: training, testing and validation.

The validation dataset is used to check the performance of our neural network during training. It is sort of the testing dataset during training.

We are using a batch of 64 initially, this means that our 22.253 training images will be grouped in 64 groups, for each of those groups a forward and backwards pass will be performed. So we will be optimizing the loss through SGD 347 times (22.253/64) for each epoch.

Images are usually fed to models using batches, groups of them, the most common sizes are 32 and 64, those numbers are chosen due to computational efficiency, batch sizes should be powers of 2, because GPU's and CPUs are also organized in powers of 2.

We are going to resize our images to 299x299. The size is chosen because InceptionV3 is trained using this size. It could be any other common size as long as it is common sense. You could also check how changing the size changes performance.

[ ]:
```
# Training dataset
train_dir = '/content/drive/MyDrive/images/train'
image_size = (299, 299)
train_ds = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    validation_split=0.1764, # % used for validation
    subset="training",
    seed=42,
    image_size=image_size, # Resize the image to 299, 299
    batch_size=64 # group them in batches of 64
)
```

```
Found 22253 files belonging to 10 classes.
Using 18328 files for training.
```

[ ]:
```
# Validation dataset
val_ds = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    validation_split=0.1764,  # % used for validation
    subset="validation",
    seed=42,
    image_size=image_size, # Resize the image to 299, 299
    batch_size=64 # group them in batches of 64
)
```

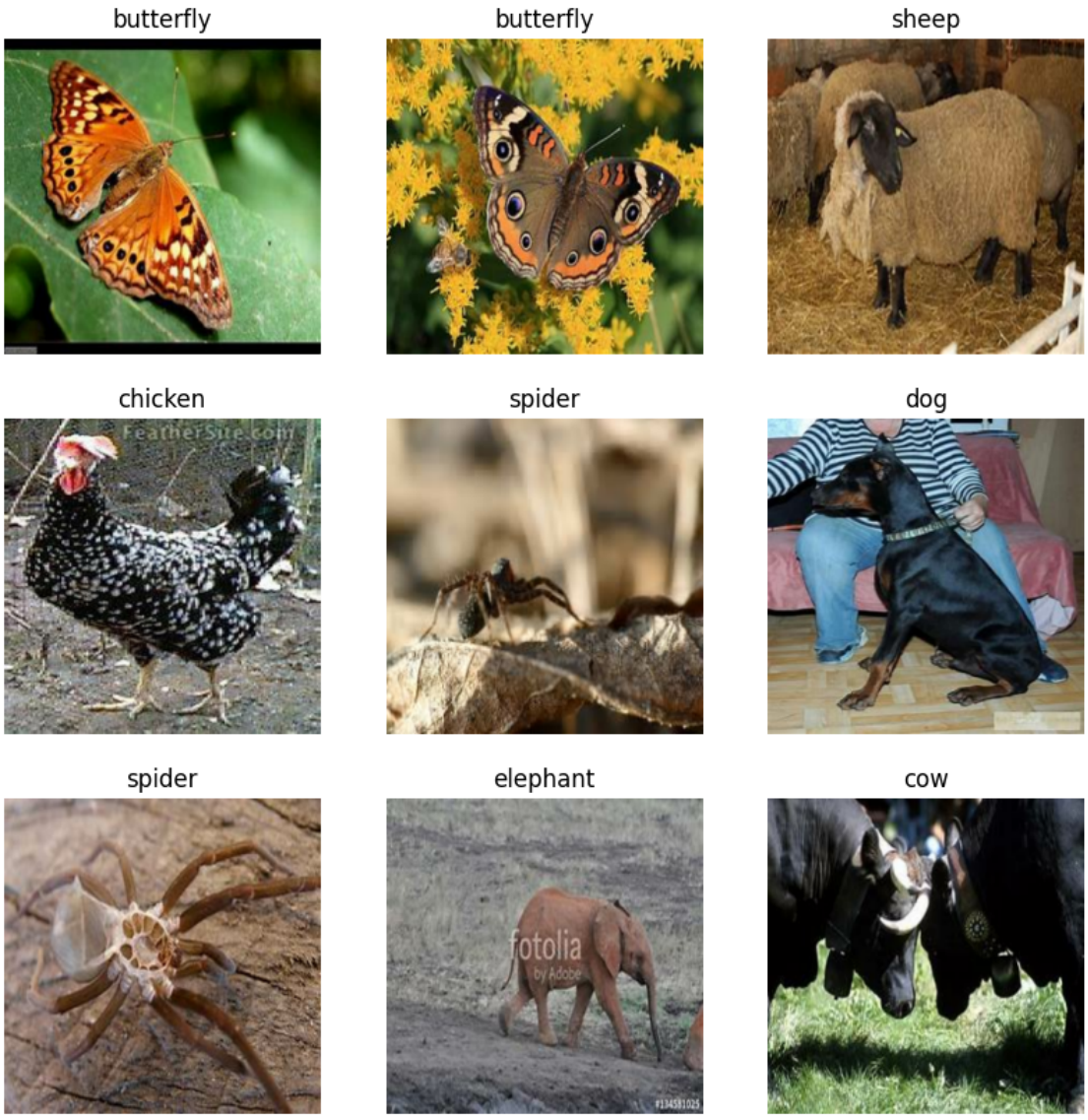
```
Found 22253 files belonging to 10 classes.
Using 3925 files for validation.
```

[ ]:
```
# Testing dataset
test_dir = '/content/drive/MyDrive/images/test'
test_ds = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    seed=42,
```

```
    image_size=image_size, # Resize the image to 299, 299
    batch_size=64 # group them in batches of 64
)
```

```
Found 3933 files belonging to 10 classes.
```

[ ]:
```
# Plotting some images and its label.
class_names = train_ds.class_names
plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```



We can check that they are all of the same size now, they have been resized.

Finally our data is structured as follows:

Training dataset: 70% Testing dataset: 15% Validation dataset: 15%

0.2 2. ANN Model

Let's first try a simple model fully connected to see what results we get.

Given that using a 299x299 image size in a fully connected neural network would mean training an excessive number of parameters, we'll define a model where firstly we resize the images to 32x32, and then we'll flatten the arrays in order to get our data in (1,3072) format. This is not necessary for convolutional neural networks, but it is mandatory for fully connected neural networks since they expect data to be in vector format.

Our model will use:

- A resizing layer from (299,299) to (32,32).
- A rescaling layer to ensure all pixel values are between 0 and 1.
- A flatten layer to transform the data into a vector of 3072 values (32x32x3).
- A fully connected layer with 2048 neurons using the relu activation function.
- A dropout layer to avoid overfitting with 0.5 probability.
- A fully connected layer with 1024 neurons and relu activation function.
- A dropout layer with a 0.5 probability.
- A fully connected layer that will act as our output layer using the soft max activation function that allows us to output probabilities where the sum of the probabilities for each class will output 1. This layer will have a neuron for each possible output class (10 in our case) and will output a probability for each one; they will sum 1 thanks to soft max.

We are using relu as the activation function because it allows us to learn either lineal and non-linear patterns, it's fast to compute, and it's the current state-of-the-art.

Just to be clear, all neural networks are artificial neural networks. Inside this category, we have:

- Fully connected neural networks, also known as feed-forward neural networks.
- Convolutional neural networks.
- Recurrent Neural Networks.
- Long short term memory networks.
- Autoencoders.
- GANs.

We are using the first type here as a simple model, and we'll move on to convolutional neural networks later on.

Feedforward needs data to be in vectors, while convolutional does not.

It's a typical structure for fully connected neural networks to have its layers in a format where we decrease a fixed % of the number of neurons as we stack layers on.

So for example first layer, 2048 neurons, second layer, 1024, third layer, 512.

This is done in order to learn low and high-level patterns, while the 2048 neurons will learn the most details, and the third layer will learn how to resume them in higher-level features.

We are going to compile and train our model using the following settings:

- Adam optimizer using a learning rate of 0.0001(1e-4). (https://towardsdatascience.com/the-math-behind-adam-optimizer-c41407efe59b) I feel like it is of the utmost importance to know how each optimizer works and what it does since it's the root of improvement for neural networks.
- 100 epochs training using Early Stopping (stop when the model does not improve) with patience of 10 epochs, we'll monitor the loss in the validation set, and we'll store the weights that gave the model the best result.
- We are going to monitor the accuracy of the training and validation datasets while training.

Finally, our loss function will be categorical cross entropy the one used for multi-classification problems.

[ ]:
```
# Model definition using Keras Sequential
def create_model(input_shape, num_classes):
    model = Sequential([
        Input(shape=input_shape),
        Resizing(32, 32),
        Rescaling(1./255),
        Flatten(),
        Dense(2048, activation='relu'),
        Dropout(0.5),
        Dense(1024, activation='relu'),
        Dropout(0.5),
        Dense(num_classes, activation='softmax')
    ])
    return model
```

[ ]:
```
# Compile the model
input_shape = (299, 299, 3)
num_classes = len(class_names)
model = create_model(input_shape, num_classes)
opt = Adam(learning_rate=1e-4)
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

early_stopping = EarlyStopping(monitor='val_loss', patience=10, verbose=1,␣
 ↪restore_best_weights=True)
```

[ ]:
```
# I'm on google colab using this GPU.
!nvidia-smi
```

```
Mon Mar 25 17:25:12 2024
+---------------------------------------------------------------------------------------
```

```
--------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version:
12.2     |
|-----------------------------------------+----------------------+----------
--------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util
Compute M. |
|                                         |                      |
MIG M. |
|=========================================+======================+==========
========|
|   0  Tesla V100-SXM2-16GB           Off | 00000000:00:04.0 Off |
0 |
| N/A   33C    P0              42W / 300W |    568MiB / 16384MiB |      0%
Default |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------+----------
--------+

+---------------------------------------------------------------------------
--------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                            GPU
Memory |
|        ID   ID
Usage      |
|===========================================================================
========|
+---------------------------------------------------------------------------
--------+
```

[ ]:
```
# Train the model
start_time = time.time()
history = model.fit(train_ds,
                    validation_data=val_ds,
                    epochs=100,
                    callbacks=[early_stopping])
end_time = time.time()
print(f"Time to train: {end_time - start_time}")
print(model.summary())
```

```
Epoch 1/100
287/287 [==============================] - 66s 228ms/step - loss: 2.0553 -
accuracy: 0.2682 - val_loss: 1.9782 - val_accuracy: 0.3210
```

```
Epoch 2/100
287/287 [==============================] - 47s 160ms/step - loss: 1.9989 -
accuracy: 0.2917 - val_loss: 1.9426 - val_accuracy: 0.3106
Epoch 3/100
287/287 [==============================] - 47s 159ms/step - loss: 1.9562 -
accuracy: 0.3060 - val_loss: 1.9192 - val_accuracy: 0.3368
Epoch 4/100
287/287 [==============================] - 46s 158ms/step - loss: 1.9276 -
accuracy: 0.3246 - val_loss: 1.9065 - val_accuracy: 0.3401
Epoch 5/100
287/287 [==============================] - 46s 157ms/step - loss: 1.9022 -
accuracy: 0.3316 - val_loss: 1.8772 - val_accuracy: 0.3450
Epoch 6/100
287/287 [==============================] - 47s 159ms/step - loss: 1.8853 -
accuracy: 0.3400 - val_loss: 1.8697 - val_accuracy: 0.3546
Epoch 7/100
287/287 [==============================] - 47s 160ms/step - loss: 1.8623 -
accuracy: 0.3506 - val_loss: 1.8482 - val_accuracy: 0.3595
Epoch 8/100
287/287 [==============================] - 47s 160ms/step - loss: 1.8371 -
accuracy: 0.3588 - val_loss: 1.8399 - val_accuracy: 0.3654
Epoch 9/100
287/287 [==============================] - 47s 159ms/step - loss: 1.8237 -
accuracy: 0.3566 - val_loss: 1.8194 - val_accuracy: 0.3643
Epoch 10/100
287/287 [==============================] - 47s 159ms/step - loss: 1.8079 -
accuracy: 0.3669 - val_loss: 1.8177 - val_accuracy: 0.3801
Epoch 11/100
287/287 [==============================] - 47s 159ms/step - loss: 1.7975 -
accuracy: 0.3739 - val_loss: 1.7954 - val_accuracy: 0.3804
Epoch 12/100
287/287 [==============================] - 46s 157ms/step - loss: 1.7809 -
accuracy: 0.3769 - val_loss: 1.7915 - val_accuracy: 0.3855
Epoch 13/100
287/287 [==============================] - 47s 160ms/step - loss: 1.7593 -
accuracy: 0.3833 - val_loss: 1.7851 - val_accuracy: 0.3804
Epoch 14/100
287/287 [==============================] - 47s 159ms/step - loss: 1.7432 -
accuracy: 0.3905 - val_loss: 1.7691 - val_accuracy: 0.3957
Epoch 15/100
287/287 [==============================] - 47s 159ms/step - loss: 1.7297 -
accuracy: 0.3978 - val_loss: 1.7679 - val_accuracy: 0.3969
Epoch 16/100
287/287 [==============================] - 47s 162ms/step - loss: 1.7169 -
accuracy: 0.4008 - val_loss: 1.7514 - val_accuracy: 0.4038
Epoch 17/100
287/287 [==============================] - 47s 161ms/step - loss: 1.7079 -
accuracy: 0.4064 - val_loss: 1.7457 - val_accuracy: 0.4041
```

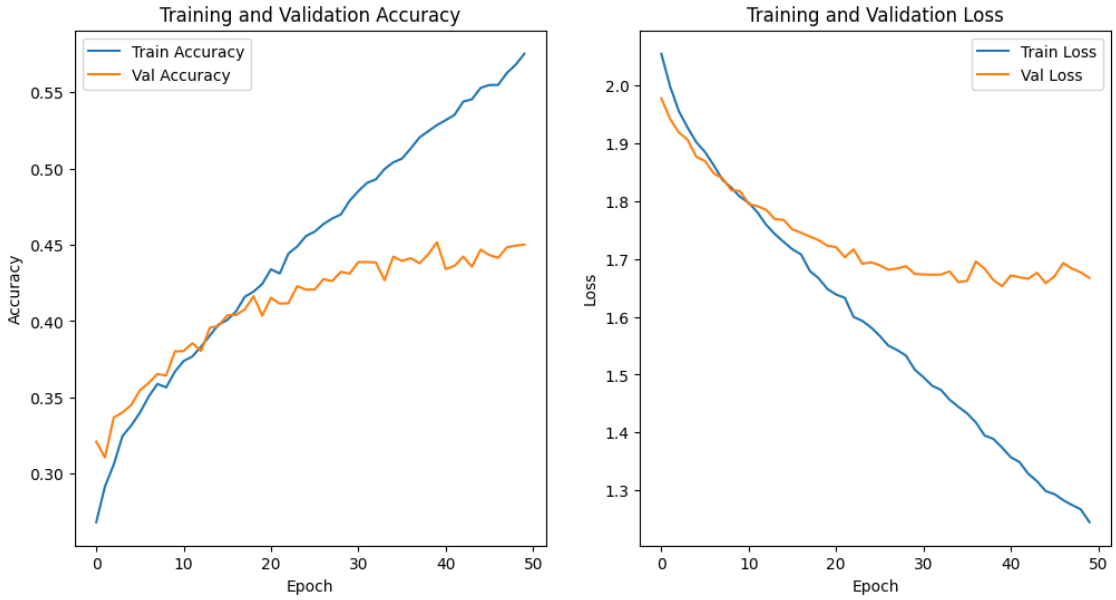
```
Epoch 18/100
287/287 [==============================] - 47s 159ms/step - loss: 1.6793 -
accuracy: 0.4160 - val_loss: 1.7390 - val_accuracy: 0.4076
Epoch 19/100
287/287 [==============================] - 46s 157ms/step - loss: 1.6668 -
accuracy: 0.4193 - val_loss: 1.7328 - val_accuracy: 0.4163
Epoch 20/100
287/287 [==============================] - 46s 158ms/step - loss: 1.6484 -
accuracy: 0.4244 - val_loss: 1.7232 - val_accuracy: 0.4036
Epoch 21/100
287/287 [==============================] - 46s 157ms/step - loss: 1.6386 -
accuracy: 0.4340 - val_loss: 1.7206 - val_accuracy: 0.4153
Epoch 22/100
287/287 [==============================] - 47s 160ms/step - loss: 1.6330 -
accuracy: 0.4312 - val_loss: 1.7031 - val_accuracy: 0.4115
Epoch 23/100
287/287 [==============================] - 47s 159ms/step - loss: 1.5999 -
accuracy: 0.4443 - val_loss: 1.7170 - val_accuracy: 0.4117
Epoch 24/100
287/287 [==============================] - 47s 159ms/step - loss: 1.5931 -
accuracy: 0.4489 - val_loss: 1.6919 - val_accuracy: 0.4229
Epoch 25/100
287/287 [==============================] - 48s 162ms/step - loss: 1.5817 -
accuracy: 0.4558 - val_loss: 1.6944 - val_accuracy: 0.4206
Epoch 26/100
287/287 [==============================] - 47s 159ms/step - loss: 1.5670 -
accuracy: 0.4587 - val_loss: 1.6894 - val_accuracy: 0.4206
Epoch 27/100
287/287 [==============================] - 46s 158ms/step - loss: 1.5500 -
accuracy: 0.4636 - val_loss: 1.6815 - val_accuracy: 0.4275
Epoch 28/100
287/287 [==============================] - 47s 159ms/step - loss: 1.5425 -
accuracy: 0.4673 - val_loss: 1.6838 - val_accuracy: 0.4262
Epoch 29/100
287/287 [==============================] - 47s 159ms/step - loss: 1.5329 -
accuracy: 0.4699 - val_loss: 1.6877 - val_accuracy: 0.4324
Epoch 30/100
287/287 [==============================] - 47s 159ms/step - loss: 1.5088 -
accuracy: 0.4789 - val_loss: 1.6744 - val_accuracy: 0.4311
Epoch 31/100
287/287 [==============================] - 46s 158ms/step - loss: 1.4955 -
accuracy: 0.4852 - val_loss: 1.6734 - val_accuracy: 0.4387
Epoch 32/100
287/287 [==============================] - 47s 160ms/step - loss: 1.4808 -
accuracy: 0.4907 - val_loss: 1.6730 - val_accuracy: 0.4387
Epoch 33/100
287/287 [==============================] - 47s 160ms/step - loss: 1.4736 -
accuracy: 0.4929 - val_loss: 1.6732 - val_accuracy: 0.4385
```

```
Epoch 34/100
287/287 [==============================] - 46s 158ms/step - loss: 1.4566 -
accuracy: 0.4997 - val_loss: 1.6786 - val_accuracy: 0.4268
Epoch 35/100
287/287 [==============================] - 47s 161ms/step - loss: 1.4443 -
accuracy: 0.5041 - val_loss: 1.6601 - val_accuracy: 0.4423
Epoch 36/100
287/287 [==============================] - 47s 161ms/step - loss: 1.4329 -
accuracy: 0.5065 - val_loss: 1.6623 - val_accuracy: 0.4395
Epoch 37/100
287/287 [==============================] - 47s 161ms/step - loss: 1.4173 -
accuracy: 0.5132 - val_loss: 1.6958 - val_accuracy: 0.4413
Epoch 38/100
287/287 [==============================] - 47s 160ms/step - loss: 1.3947 -
accuracy: 0.5204 - val_loss: 1.6833 - val_accuracy: 0.4380
Epoch 39/100
287/287 [==============================] - 47s 161ms/step - loss: 1.3888 -
accuracy: 0.5245 - val_loss: 1.6637 - val_accuracy: 0.4438
Epoch 40/100
287/287 [==============================] - 47s 160ms/step - loss: 1.3737 -
accuracy: 0.5285 - val_loss: 1.6530 - val_accuracy: 0.4517
Epoch 41/100
287/287 [==============================] - 47s 160ms/step - loss: 1.3571 -
accuracy: 0.5318 - val_loss: 1.6716 - val_accuracy: 0.4341
Epoch 42/100
287/287 [==============================] - 47s 159ms/step - loss: 1.3486 -
accuracy: 0.5352 - val_loss: 1.6686 - val_accuracy: 0.4362
Epoch 43/100
287/287 [==============================] - 47s 160ms/step - loss: 1.3284 -
accuracy: 0.5439 - val_loss: 1.6660 - val_accuracy: 0.4423
Epoch 44/100
287/287 [==============================] - 47s 160ms/step - loss: 1.3155 -
accuracy: 0.5454 - val_loss: 1.6763 - val_accuracy: 0.4357
Epoch 45/100
287/287 [==============================] - 47s 160ms/step - loss: 1.2983 -
accuracy: 0.5529 - val_loss: 1.6582 - val_accuracy: 0.4469
Epoch 46/100
287/287 [==============================] - 47s 159ms/step - loss: 1.2932 -
accuracy: 0.5548 - val_loss: 1.6702 - val_accuracy: 0.4433
Epoch 47/100
287/287 [==============================] - 47s 161ms/step - loss: 1.2828 -
accuracy: 0.5548 - val_loss: 1.6930 - val_accuracy: 0.4415
Epoch 48/100
287/287 [==============================] - 47s 161ms/step - loss: 1.2745 -
accuracy: 0.5627 - val_loss: 1.6835 - val_accuracy: 0.4484
Epoch 49/100
287/287 [==============================] - 47s 160ms/step - loss: 1.2668 -
accuracy: 0.5680 - val_loss: 1.6770 - val_accuracy: 0.4494
```

```
Epoch 50/100
286/287 [============================>.] - ETA: 0s - loss: 1.2450 - accuracy:
0.5751Restoring model weights from the end of the best epoch: 40.
287/287 [==============================] - 47s 160ms/step - loss: 1.2447 -
accuracy: 0.5752 - val_loss: 1.6674 - val_accuracy: 0.4502
Epoch 50: early stopping
Temps d'entrenament: 2394.5640528202057
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resizing (Resizing)         (None, 32, 32, 3)         0

 rescaling (Rescaling)       (None, 32, 32, 3)         0

 flatten (Flatten)           (None, 3072)              0

 dense (Dense)               (None, 2048)              6293504

 dropout (Dropout)           (None, 2048)              0

 dense_1 (Dense)             (None, 1024)              2098176

 dropout_1 (Dropout)         (None, 1024)              0

 dense_2 (Dense)             (None, 10)                10250

=================================================================
Total params: 8401930 (32.05 MB)
Trainable params: 8401930 (32.05 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

[ ]:
```python
# Function to represent graphically the progress of accuracy and loss
# for the training and testing datasets, we'll look for overfitting
# looking at these graphs.
def plot_history(history):
    plt.figure(figsize=(12, 6))

    # Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Val Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
```

```
    plt.legend()

    # Loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.show()

plot_history(history)
```



[ ]:
```
# Evaluate the model on test data
loss, accuracy = model.evaluate(test_ds)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)
```

```
62/62 [==============================] - 446s 6s/step - loss: 1.7722 - accuracy:
0.3994
Test Loss: 1.7721571922302246
Test Accuracy: 0.39944061636924744
```

The structure and number of neurons per layer in our model are hyperparameters, meaning that they can be changed and are fixed before training.

The ideal choice is found through hyperparameter tuning, which means that we'll train and test many models with many structures and different numbers of neurons, and we'll choose the one that performs the best. This requires more computing power than you'll be able to get in a Jupyter notebook, and it's out of our scope but can be easily done on cloud platforms.

Our model has 8.401.930 parameters or weights that are trainable; imagine if we had not reduced the images from 299x299 to 32x32. Our training time is about 40 minutes.

The accuracy achieved by the model on the test set is not a good result, 40%, yet taking into account that if we guessed randomly, we would get approximately 10% accuracy, we can easily see that at least it seems the model learned something.

If we take a look at the accuracy and loss curves, we can see a clear overfitting case; the curves of the validation set flatten much earlier than the training ones, meaning our model is learning about the training data, but when it applies what it has learned the validation set, the results are poor, so we could say the model is not grasping a good generalization of the data structure.

Our training process is stopped once we have not bettered the result on the validation set for 10 epochs.

We'll need to construct a model that is able to abstract more patterns and generalize relations better than this one does.

Images are not good stuff to represent as vectors. The reason is that is more important the context in which a pixel is situated than the value of the pixel itself. We need to abstract zones of the images and get to a higher level than studying the pixel values by itself.

So we'll now move on to convolutional layers and maxpoolayers, that will enable us to combine pixel values and combine them into zones, and these zones into even bigger zones.

Convolutional layers are made for data in multidimensional format, and they are best performing there; that's why we'll use them.

Convolution explained: https://en.wikipedia.org/wiki/Convolution MaxPooling2d: https://keras.io/api/layers/pooling_layers/max_pooling2d/

0.3 3. Modest convolutional neural network.

Given the lackluster performance of the previous model, we'll try CNN's.

They are specifically good at modeling data in two dimensions, as in our images.

The CNN structure is divided into two blocks:

- Feature extractor: This block generates the different abstraction levels. The deeper those layers are, the better the performance, at least to a certain extent.
- Classifier: This block will use fully connected layers, and the output will be the probability associated with each class.

In our previous model, the feature extractor was very simple; we just took the pixel values.

We will now use convolutional layers to learn better abstractions of the images.

Our feature extractor will use:

- A rescaling input layer, will take images of size (299, 299, 3) and transform the dynamic range of the pixel values to be between 0 and 1 instead of 0 and 255.
- 3 convolutional layers with a kernel (3x3), padding ='same' and activation function relu, we will use 16, 32 and 64 neurons for each layer, going from down to up.
- Each of those layers will be followed by a MaxPooling2D layer, which will feed higher-level data to the next layer.
- A dropout layer with 0.2 probability to avoid overfitting.

As you can see, we have now reversed the pattern used to construct layers previously; instead of 64, 32, 16, we are doing 16, 32, 64.

What we are doing is dedicating a higher number of neurons to the convolutional layers that will deal with the higher-level data.

You can see that we are using a MaxPool2D layer after each convolution. Maxpooling takes a range of pixels determined by the pool_size parameter and resumes that zone into one value, the maximum value of that zone, and does that all along the image for each zone fittable in the pool_size.

This means that after applying this layer, the information we are feeding the next convolutional neural network will be of a higher level, a resumed version of the previous data.

So we are dedicating more neurons to the higher-level information, so our feature extractor is giving more weight to the higher-level information than the raw pixel information, although it also has its weight. Thus, we are able to learn a better generalization.

Our classifier will use:

- A flatten layer to obtain a vector, instead of 2D data, from the resulting information provided by the feature extractor.
- A fully connected layer with 64 neurons and relu as activation function.
- A dropout layer with a 0.5 probability.
- A fully connected output layer, the number of neurons is equal to the number of possible classes (in our case, 10), and the activation function will be soft max to guarantee the sum of output probabilities adds up to 1.

As you can see, we flatten our data now. Can you see why? The classifier is a fully connected layer, just as was our previous model, so basically we have just used a block of convolutional and max pooling layers to abstract the data, and then we pass that processed data to a fully connected model to classify it.

We are going to compile and train our model with the following settings:

- Adam optimizer using a learning rate of 0.0001(1e-4).
- 100 epochs training using Early Stopping and patience of 10 epochs, we will monitor the validation set loss function, and we will keep the weights of the best model,.
- ReduceLRONPlateau monitoring the loss function of the validation set, using patience of 5 epochs, a factor of 0.2, and a learning rate of 0.000001(1e-6).
- We are going to monitor the accuracy and loss of both datasets during training.

https://keras.io/api/callbacks/reduce_lr_on_plateau/ Reduce LR on Plateau will reduce the learning rate if, in 5 epochs, there has been no progress in the validation loss. This is done in

order to try to revive the training when, in 5 epochs, it has not been improving. This may be happening because the learning rate is high enough to simply overpass the minimum point of the loss function, so we try to take a shorter step and see if there is a minimum there that can improve our loss function before stopping our training.

The loss function keeps being the categorical cross entropy, as the nature of the problem has not changed.

[ ]:
```
# Model definition
CNN = Sequential([
    # Feature extractor
    Rescaling(1./255, input_shape=(299, 299, 3)),
    Conv2D(16, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(32, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.2),
    # Classifier
    Flatten(),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(len(class_names), activation='softmax')
])
```

[ ]:
```
# Compiling
CNN.compile(optimizer=Adam(learning_rate=0.0001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

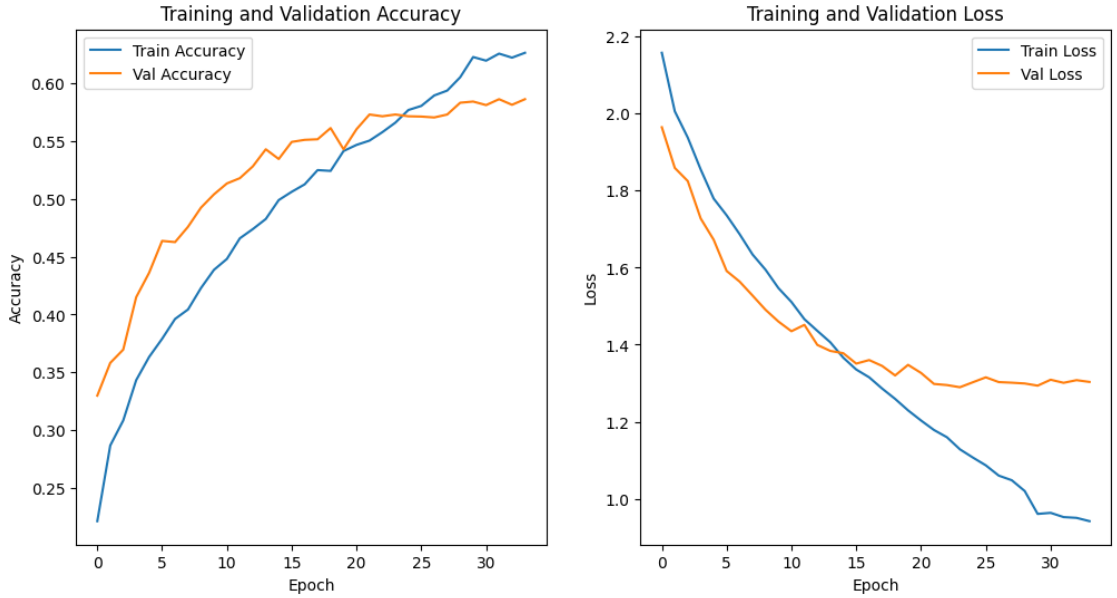
[ ]:
```
# Training
start_time = time.time()
history = CNN.fit(train_ds,
                  epochs=100,
                  validation_data=val_ds,
                  callbacks=[
                      EarlyStopping(patience=10, monitor='val_loss',␣
 ↪restore_best_weights=True),
                      ReduceLROnPlateau(patience=5, factor=0.2, min_lr=1e-6,␣
 ↪monitor='val_loss', verbose=1)
                  ])
final_time = time.time()
plot_history(history)
```

```
Epoch 1/100
287/287 [==============================] - 53s 164ms/step - loss: 2.1581 -
accuracy: 0.2214 - val_loss: 1.9648 - val_accuracy: 0.3299 - lr: 1.0000e-04
```

```
Epoch 2/100
287/287 [==============================] - 47s 159ms/step - loss: 2.0062 -
accuracy: 0.2868 - val_loss: 1.8592 - val_accuracy: 0.3582 - lr: 1.0000e-04
Epoch 3/100
287/287 [==============================] - 47s 160ms/step - loss: 1.9379 -
accuracy: 0.3084 - val_loss: 1.8250 - val_accuracy: 0.3697 - lr: 1.0000e-04
Epoch 4/100
287/287 [==============================] - 48s 163ms/step - loss: 1.8550 -
accuracy: 0.3433 - val_loss: 1.7277 - val_accuracy: 0.4150 - lr: 1.0000e-04
Epoch 5/100
287/287 [==============================] - 47s 161ms/step - loss: 1.7794 -
accuracy: 0.3634 - val_loss: 1.6726 - val_accuracy: 0.4362 - lr: 1.0000e-04
Epoch 6/100
287/287 [==============================] - 48s 163ms/step - loss: 1.7360 -
accuracy: 0.3790 - val_loss: 1.5915 - val_accuracy: 0.4637 - lr: 1.0000e-04
Epoch 7/100
287/287 [==============================] - 47s 162ms/step - loss: 1.6872 -
accuracy: 0.3962 - val_loss: 1.5640 - val_accuracy: 0.4627 - lr: 1.0000e-04
Epoch 8/100
287/287 [==============================] - 47s 162ms/step - loss: 1.6343 -
accuracy: 0.4044 - val_loss: 1.5277 - val_accuracy: 0.4759 - lr: 1.0000e-04
Epoch 9/100
287/287 [==============================] - 47s 161ms/step - loss: 1.5948 -
accuracy: 0.4230 - val_loss: 1.4907 - val_accuracy: 0.4925 - lr: 1.0000e-04
Epoch 10/100
287/287 [==============================] - 47s 161ms/step - loss: 1.5469 -
accuracy: 0.4387 - val_loss: 1.4601 - val_accuracy: 0.5039 - lr: 1.0000e-04
Epoch 11/100
287/287 [==============================] - 48s 162ms/step - loss: 1.5110 -
accuracy: 0.4481 - val_loss: 1.4350 - val_accuracy: 0.5134 - lr: 1.0000e-04
Epoch 12/100
287/287 [==============================] - 47s 161ms/step - loss: 1.4663 -
accuracy: 0.4660 - val_loss: 1.4516 - val_accuracy: 0.5180 - lr: 1.0000e-04
Epoch 13/100
287/287 [==============================] - 48s 163ms/step - loss: 1.4360 -
accuracy: 0.4739 - val_loss: 1.3994 - val_accuracy: 0.5282 - lr: 1.0000e-04
Epoch 14/100
287/287 [==============================] - 48s 163ms/step - loss: 1.4063 -
accuracy: 0.4826 - val_loss: 1.3839 - val_accuracy: 0.5429 - lr: 1.0000e-04
Epoch 15/100
287/287 [==============================] - 47s 161ms/step - loss: 1.3663 -
accuracy: 0.4990 - val_loss: 1.3780 - val_accuracy: 0.5345 - lr: 1.0000e-04
Epoch 16/100
287/287 [==============================] - 47s 161ms/step - loss: 1.3356 -
accuracy: 0.5062 - val_loss: 1.3509 - val_accuracy: 0.5493 - lr: 1.0000e-04
Epoch 17/100
287/287 [==============================] - 47s 162ms/step - loss: 1.3153 -
accuracy: 0.5126 - val_loss: 1.3601 - val_accuracy: 0.5511 - lr: 1.0000e-04
```

```
Epoch 18/100
287/287 [==============================] - 47s 162ms/step - loss: 1.2861 -
accuracy: 0.5249 - val_loss: 1.3449 - val_accuracy: 0.5516 - lr: 1.0000e-04
Epoch 19/100
287/287 [==============================] - 48s 162ms/step - loss: 1.2593 -
accuracy: 0.5242 - val_loss: 1.3202 - val_accuracy: 0.5613 - lr: 1.0000e-04
Epoch 20/100
287/287 [==============================] - 47s 162ms/step - loss: 1.2295 -
accuracy: 0.5414 - val_loss: 1.3477 - val_accuracy: 0.5429 - lr: 1.0000e-04
Epoch 21/100
287/287 [==============================] - 48s 162ms/step - loss: 1.2032 -
accuracy: 0.5467 - val_loss: 1.3268 - val_accuracy: 0.5603 - lr: 1.0000e-04
Epoch 22/100
287/287 [==============================] - 47s 162ms/step - loss: 1.1785 -
accuracy: 0.5504 - val_loss: 1.2981 - val_accuracy: 0.5730 - lr: 1.0000e-04
Epoch 23/100
287/287 [==============================] - 48s 164ms/step - loss: 1.1597 -
accuracy: 0.5578 - val_loss: 1.2952 - val_accuracy: 0.5715 - lr: 1.0000e-04
Epoch 24/100
287/287 [==============================] - 48s 163ms/step - loss: 1.1284 -
accuracy: 0.5660 - val_loss: 1.2896 - val_accuracy: 0.5730 - lr: 1.0000e-04
Epoch 25/100
287/287 [==============================] - 48s 164ms/step - loss: 1.1070 -
accuracy: 0.5769 - val_loss: 1.3025 - val_accuracy: 0.5715 - lr: 1.0000e-04
Epoch 26/100
287/287 [==============================] - 47s 160ms/step - loss: 1.0867 -
accuracy: 0.5804 - val_loss: 1.3153 - val_accuracy: 0.5712 - lr: 1.0000e-04
Epoch 27/100
287/287 [==============================] - 47s 160ms/step - loss: 1.0600 -
accuracy: 0.5895 - val_loss: 1.3029 - val_accuracy: 0.5704 - lr: 1.0000e-04
Epoch 28/100
287/287 [==============================] - 47s 161ms/step - loss: 1.0481 -
accuracy: 0.5937 - val_loss: 1.3014 - val_accuracy: 0.5730 - lr: 1.0000e-04
Epoch 29/100
286/287 [============================>.] - ETA: 0s - loss: 1.0203 - accuracy:
0.6052
Epoch 29: ReduceLROnPlateau reducing learning rate to 1.9999999494757503e-05.
287/287 [==============================] - 48s 163ms/step - loss: 1.0201 -
accuracy: 0.6052 - val_loss: 1.2993 - val_accuracy: 0.5832 - lr: 1.0000e-04
Epoch 30/100
287/287 [==============================] - 47s 161ms/step - loss: 0.9606 -
accuracy: 0.6227 - val_loss: 1.2937 - val_accuracy: 0.5842 - lr: 2.0000e-05
Epoch 31/100
287/287 [==============================] - 48s 164ms/step - loss: 0.9634 -
accuracy: 0.6195 - val_loss: 1.3090 - val_accuracy: 0.5811 - lr: 2.0000e-05
Epoch 32/100
287/287 [==============================] - 48s 163ms/step - loss: 0.9523 -
accuracy: 0.6257 - val_loss: 1.3011 - val_accuracy: 0.5862 - lr: 2.0000e-05
```

```
Epoch 33/100
287/287 [==============================] - 48s 165ms/step - loss: 0.9503 -
accuracy: 0.6222 - val_loss: 1.3079 - val_accuracy: 0.5814 - lr: 2.0000e-05
Epoch 34/100
286/287 [============================>.] - ETA: 0s - loss: 0.9415 - accuracy:
0.6266
Epoch 34: ReduceLROnPlateau reducing learning rate to 3.999999898951501e-06.
287/287 [==============================] - 48s 162ms/step - loss: 0.9418 -
accuracy: 0.6264 - val_loss: 1.3033 - val_accuracy: 0.5862 - lr: 2.0000e-05
```



[ ]:
```python
# Evaluate the model against the test dataset
loss, accuracy = CNN.evaluate(test_ds)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)
```

```
62/62 [==============================] - 8s 116ms/step - loss: 1.3669 -
accuracy: 0.5477
Pèrdua final sobre les dades de test: 1.3669072389602661
Exactitud final sobre les dades de test: 0.5476735234260559
```

[ ]:
```python
total_params = np.sum([np.prod(w.shape) for w in CNN.trainable_weights])
print("Number of trainable parameters:", total_params)
print(f"Training time: {final_time - start_time}")
print(f"The accuracy achieven by the model is {accuracy}")
```

```
Nombre de paràmetres a entrenar: 5631722
Temps d'entrenament: 1725.3137938976288
L'accuracy sobre el conjunt de test obtinguda pel model és 0.5476735234260559
```

We have improved the previous result from 40% to 55%, yet this result does not satisfy us.

Overfitting is also still there; the validation curves flatten much earlier than the training ones, yet it is not as evident as in the previous model.

We are going to introduce a new element that is commonly used in computer vision problems to help the models generalize better. We are going to augmentate our data.

Basically randomly flipping, rotating or zooming our images, so the model focuses on a high level abstraction of what is a spider instead of specific details.

Let's try it.

0.4 4. Data Augmentation

One way to solve our problems would be to acquire more data. Since this is not always possible, we will augmentate our data by slightly modifying it, so our model focuses more on the patterns than the specific details.

[ ]:
```
# Define the augmentation model
augmentation_model = Sequential([
    RandomFlip("horizontal", input_shape=image_size + (3,)),
    RandomRotation(0.1)
])
```

[ ]:
```
# Plot some images after augmenting them
import numpy
for images, _ in train_ds.take(1):
    sample_images = images
    break

# Pass the images through the model
output_images = augmentation_model(sample_images, training=True)

# Plot them
plt.figure(figsize=(10, 5))
for i in range(4):
    plt.subplot(2, 4, i + 1)
    plt.imshow(sample_images[i].numpy().astype("uint8"))
    plt.axis('off')
    plt.title('Original image')

    plt.subplot(2, 4, i + 5)
    plt.imshow(output_images[i].numpy().astype("uint8"))
    plt.axis('off')
    plt.title('Processed image')

plt.show()
```

We will compile and train the model using the follow settings: - Adam optimizer with a learning rate of 0.001(1e-3), we want to learn faster as the data has been augmentated. - Train for 100 epochs using EarlyStopping with patience of 10 epochs, monitoring the loss function of the validation set, and keeping the weights of the best model. - Use ReduceLROnPlateau to monitor the loss function in the validation set, with patience of 5 epochs, a factor of 0.2, and a learning rate of 0.0000001(1e-6). - Monitoring accuracy and loss during training and validation.

[ ]:
```
# Model definition
CNN_augmented = Sequential([
    # Feature extractor
    Rescaling(1./255, input_shape=(299, 299, 3)),
    ## Data augmentation
    RandomFlip("horizontal", input_shape=(299, 299, 3)),
    RandomRotation(0.1),
    ## Convolutional layers
    Conv2D(16, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(32, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.2),
    # Classifier
    Flatten(),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(len(class_names), activation='softmax')
])
```

[ ]:
```
# Compile the model
CNN_augmented.compile(optimizer=Adam(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

[ ]:
```
# Training
start_time = time.time()
history = CNN_augmented.fit(train_ds,
                    epochs=100,
                    validation_data=val_ds,
                    callbacks=[
                        EarlyStopping(patience=10, monitor='val_loss',␣
 ↪restore_best_weights=True),
                        ReduceLROnPlateau(patience=5, factor=0.2, min_lr=1e-6,␣
 ↪monitor='val_loss', verbose=1)
                    ])
final_time = time.time()
plot_history(history)
```

```
Epoch 1/100
287/287 [==============================] - 51s 166ms/step - loss: 2.1218 -
accuracy: 0.2430 - val_loss: 1.9463 - val_accuracy: 0.3414 - lr: 0.0010
Epoch 2/100
287/287 [==============================] - 48s 162ms/step - loss: 1.9445 -
accuracy: 0.3198 - val_loss: 1.7672 - val_accuracy: 0.3985 - lr: 0.0010
Epoch 3/100
287/287 [==============================] - 48s 162ms/step - loss: 1.8410 -
accuracy: 0.3471 - val_loss: 1.6775 - val_accuracy: 0.4224 - lr: 0.0010
Epoch 4/100
287/287 [==============================] - 48s 163ms/step - loss: 1.7985 -
accuracy: 0.3597 - val_loss: 1.5716 - val_accuracy: 0.4489 - lr: 0.0010
Epoch 5/100
287/287 [==============================] - 48s 163ms/step - loss: 1.7581 -
accuracy: 0.3708 - val_loss: 1.5789 - val_accuracy: 0.4535 - lr: 0.0010
Epoch 6/100
287/287 [==============================] - 48s 165ms/step - loss: 1.7167 -
accuracy: 0.3915 - val_loss: 1.5124 - val_accuracy: 0.4762 - lr: 0.0010
Epoch 7/100
287/287 [==============================] - 49s 168ms/step - loss: 1.6731 -
accuracy: 0.4002 - val_loss: 1.4836 - val_accuracy: 0.4795 - lr: 0.0010
Epoch 8/100
287/287 [==============================] - 49s 166ms/step - loss: 1.6496 -
accuracy: 0.4070 - val_loss: 1.5123 - val_accuracy: 0.4701 - lr: 0.0010
Epoch 9/100
287/287 [==============================] - 49s 166ms/step - loss: 1.6194 -
accuracy: 0.4202 - val_loss: 1.4383 - val_accuracy: 0.4917 - lr: 0.0010
Epoch 10/100
```

```
287/287 [==============================] - 50s 170ms/step - loss: 1.5977 -
accuracy: 0.4187 - val_loss: 1.4081 - val_accuracy: 0.5062 - lr: 0.0010
Epoch 11/100
287/287 [==============================] - 49s 166ms/step - loss: 1.5907 -
accuracy: 0.4280 - val_loss: 1.3878 - val_accuracy: 0.5073 - lr: 0.0010
Epoch 12/100
287/287 [==============================] - 48s 165ms/step - loss: 1.5637 -
accuracy: 0.4328 - val_loss: 1.3812 - val_accuracy: 0.5106 - lr: 0.0010
Epoch 13/100
287/287 [==============================] - 49s 167ms/step - loss: 1.5610 -
accuracy: 0.4359 - val_loss: 1.3225 - val_accuracy: 0.5332 - lr: 0.0010
Epoch 14/100
287/287 [==============================] - 49s 167ms/step - loss: 1.5251 -
accuracy: 0.4468 - val_loss: 1.3078 - val_accuracy: 0.5401 - lr: 0.0010
Epoch 15/100
287/287 [==============================] - 49s 167ms/step - loss: 1.5257 -
accuracy: 0.4422 - val_loss: 1.3291 - val_accuracy: 0.5322 - lr: 0.0010
Epoch 16/100
287/287 [==============================] - 49s 167ms/step - loss: 1.4965 -
accuracy: 0.4549 - val_loss: 1.2590 - val_accuracy: 0.5526 - lr: 0.0010
Epoch 17/100
287/287 [==============================] - 49s 165ms/step - loss: 1.4881 -
accuracy: 0.4590 - val_loss: 1.2424 - val_accuracy: 0.5546 - lr: 0.0010
Epoch 18/100
287/287 [==============================] - 49s 167ms/step - loss: 1.4727 -
accuracy: 0.4614 - val_loss: 1.2407 - val_accuracy: 0.5783 - lr: 0.0010
Epoch 19/100
287/287 [==============================] - 49s 166ms/step - loss: 1.4573 -
accuracy: 0.4693 - val_loss: 1.3365 - val_accuracy: 0.5315 - lr: 0.0010
Epoch 20/100
287/287 [==============================] - 49s 166ms/step - loss: 1.4412 -
accuracy: 0.4744 - val_loss: 1.2308 - val_accuracy: 0.5671 - lr: 0.0010
Epoch 21/100
287/287 [==============================] - 48s 164ms/step - loss: 1.4314 -
accuracy: 0.4800 - val_loss: 1.2603 - val_accuracy: 0.5791 - lr: 0.0010
Epoch 22/100
287/287 [==============================] - 49s 166ms/step - loss: 1.4036 -
accuracy: 0.4917 - val_loss: 1.2092 - val_accuracy: 0.5944 - lr: 0.0010
Epoch 23/100
287/287 [==============================] - 49s 165ms/step - loss: 1.3801 -
accuracy: 0.5051 - val_loss: 1.1798 - val_accuracy: 0.6008 - lr: 0.0010
Epoch 24/100
287/287 [==============================] - 49s 166ms/step - loss: 1.3785 -
accuracy: 0.5004 - val_loss: 1.2050 - val_accuracy: 0.5941 - lr: 0.0010
Epoch 25/100
287/287 [==============================] - 49s 168ms/step - loss: 1.3640 -
accuracy: 0.5066 - val_loss: 1.1687 - val_accuracy: 0.6064 - lr: 0.0010
Epoch 26/100
```

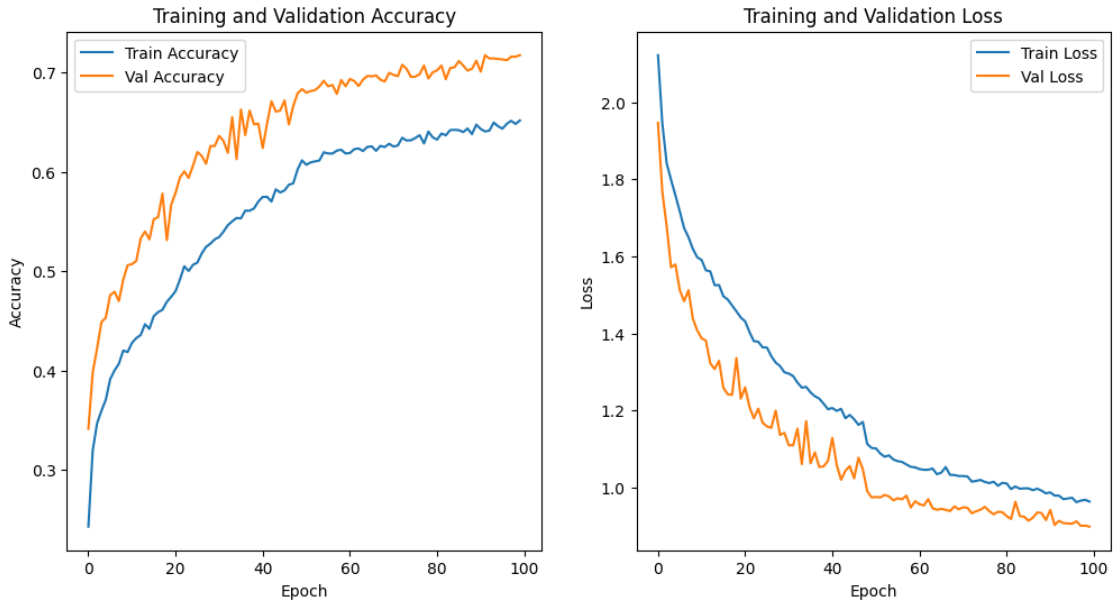
```
287/287 [==============================] - 49s 165ms/step - loss: 1.3635 -
accuracy: 0.5088 - val_loss: 1.1590 - val_accuracy: 0.6201 - lr: 0.0010
Epoch 27/100
287/287 [==============================] - 48s 163ms/step - loss: 1.3410 -
accuracy: 0.5180 - val_loss: 1.1552 - val_accuracy: 0.6163 - lr: 0.0010
Epoch 28/100
287/287 [==============================] - 49s 165ms/step - loss: 1.3246 -
accuracy: 0.5247 - val_loss: 1.1998 - val_accuracy: 0.6084 - lr: 0.0010
Epoch 29/100
287/287 [==============================] - 48s 163ms/step - loss: 1.3155 -
accuracy: 0.5279 - val_loss: 1.1367 - val_accuracy: 0.6262 - lr: 0.0010
Epoch 30/100
287/287 [==============================] - 48s 165ms/step - loss: 1.2994 -
accuracy: 0.5323 - val_loss: 1.1422 - val_accuracy: 0.6262 - lr: 0.0010
Epoch 31/100
287/287 [==============================] - 48s 164ms/step - loss: 1.2959 -
accuracy: 0.5345 - val_loss: 1.1104 - val_accuracy: 0.6364 - lr: 0.0010
Epoch 32/100
287/287 [==============================] - 49s 167ms/step - loss: 1.2895 -
accuracy: 0.5400 - val_loss: 1.1097 - val_accuracy: 0.6313 - lr: 0.0010
Epoch 33/100
287/287 [==============================] - 49s 165ms/step - loss: 1.2727 -
accuracy: 0.5465 - val_loss: 1.1531 - val_accuracy: 0.6194 - lr: 0.0010
Epoch 34/100
287/287 [==============================] - 49s 167ms/step - loss: 1.2594 -
accuracy: 0.5504 - val_loss: 1.0613 - val_accuracy: 0.6553 - lr: 0.0010
Epoch 35/100
287/287 [==============================] - 49s 165ms/step - loss: 1.2614 -
accuracy: 0.5538 - val_loss: 1.1726 - val_accuracy: 0.6130 - lr: 0.0010
Epoch 36/100
287/287 [==============================] - 48s 165ms/step - loss: 1.2471 -
accuracy: 0.5534 - val_loss: 1.0635 - val_accuracy: 0.6629 - lr: 0.0010
Epoch 37/100
287/287 [==============================] - 47s 162ms/step - loss: 1.2371 -
accuracy: 0.5612 - val_loss: 1.0916 - val_accuracy: 0.6372 - lr: 0.0010
Epoch 38/100
287/287 [==============================] - 47s 161ms/step - loss: 1.2310 -
accuracy: 0.5611 - val_loss: 1.0538 - val_accuracy: 0.6619 - lr: 0.0010
Epoch 39/100
287/287 [==============================] - 48s 162ms/step - loss: 1.2179 -
accuracy: 0.5634 - val_loss: 1.0553 - val_accuracy: 0.6482 - lr: 0.0010
Epoch 40/100
287/287 [==============================] - 48s 163ms/step - loss: 1.2035 -
accuracy: 0.5704 - val_loss: 1.0697 - val_accuracy: 0.6487 - lr: 0.0010
Epoch 41/100
287/287 [==============================] - 47s 162ms/step - loss: 1.2067 -
accuracy: 0.5750 - val_loss: 1.1293 - val_accuracy: 0.6242 - lr: 0.0010
Epoch 42/100
```

```
287/287 [==============================] - 47s 161ms/step - loss: 1.1991 -
accuracy: 0.5750 - val_loss: 1.0592 - val_accuracy: 0.6492 - lr: 0.0010
Epoch 43/100
287/287 [==============================] - 47s 161ms/step - loss: 1.2045 -
accuracy: 0.5703 - val_loss: 1.0207 - val_accuracy: 0.6713 - lr: 0.0010
Epoch 44/100
287/287 [==============================] - 48s 162ms/step - loss: 1.1801 -
accuracy: 0.5826 - val_loss: 1.0439 - val_accuracy: 0.6609 - lr: 0.0010
Epoch 45/100
287/287 [==============================] - 48s 162ms/step - loss: 1.1886 -
accuracy: 0.5795 - val_loss: 1.0561 - val_accuracy: 0.6619 - lr: 0.0010
Epoch 46/100
287/287 [==============================] - 48s 163ms/step - loss: 1.1776 -
accuracy: 0.5815 - val_loss: 1.0243 - val_accuracy: 0.6721 - lr: 0.0010
Epoch 47/100
287/287 [==============================] - 48s 164ms/step - loss: 1.1628 -
accuracy: 0.5873 - val_loss: 1.0777 - val_accuracy: 0.6479 - lr: 0.0010
Epoch 48/100
286/287 [============================>.] - ETA: 0s - loss: 1.1703 - accuracy:
0.5887
Epoch 48: ReduceLROnPlateau reducing learning rate to 0.00020000000949949026.
287/287 [==============================] - 48s 165ms/step - loss: 1.1704 -
accuracy: 0.5886 - val_loss: 1.0477 - val_accuracy: 0.6665 - lr: 0.0010
Epoch 49/100
287/287 [==============================] - 48s 163ms/step - loss: 1.1135 -
accuracy: 0.6026 - val_loss: 0.9912 - val_accuracy: 0.6792 - lr: 2.0000e-04
Epoch 50/100
287/287 [==============================] - 48s 163ms/step - loss: 1.1031 -
accuracy: 0.6118 - val_loss: 0.9750 - val_accuracy: 0.6836 - lr: 2.0000e-04
Epoch 51/100
287/287 [==============================] - 48s 163ms/step - loss: 1.1019 -
accuracy: 0.6073 - val_loss: 0.9759 - val_accuracy: 0.6800 - lr: 2.0000e-04
Epoch 52/100
287/287 [==============================] - 47s 162ms/step - loss: 1.0887 -
accuracy: 0.6098 - val_loss: 0.9747 - val_accuracy: 0.6815 - lr: 2.0000e-04
Epoch 53/100
287/287 [==============================] - 48s 163ms/step - loss: 1.0803 -
accuracy: 0.6108 - val_loss: 0.9813 - val_accuracy: 0.6825 - lr: 2.0000e-04
Epoch 54/100
287/287 [==============================] - 48s 165ms/step - loss: 1.0838 -
accuracy: 0.6116 - val_loss: 0.9773 - val_accuracy: 0.6861 - lr: 2.0000e-04
Epoch 55/100
287/287 [==============================] - 48s 163ms/step - loss: 1.0732 -
accuracy: 0.6199 - val_loss: 0.9674 - val_accuracy: 0.6920 - lr: 2.0000e-04
Epoch 56/100
287/287 [==============================] - 48s 163ms/step - loss: 1.0688 -
accuracy: 0.6187 - val_loss: 0.9724 - val_accuracy: 0.6864 - lr: 2.0000e-04
Epoch 57/100
```

```
287/287 [==============================] - 47s 162ms/step - loss: 1.0667 -
accuracy: 0.6186 - val_loss: 0.9699 - val_accuracy: 0.6876 - lr: 2.0000e-04
Epoch 58/100
287/287 [==============================] - 47s 160ms/step - loss: 1.0601 -
accuracy: 0.6216 - val_loss: 0.9792 - val_accuracy: 0.6787 - lr: 2.0000e-04
Epoch 59/100
287/287 [==============================] - 47s 161ms/step - loss: 1.0542 -
accuracy: 0.6225 - val_loss: 0.9486 - val_accuracy: 0.6927 - lr: 2.0000e-04
Epoch 60/100
287/287 [==============================] - 48s 163ms/step - loss: 1.0526 -
accuracy: 0.6187 - val_loss: 0.9649 - val_accuracy: 0.6864 - lr: 2.0000e-04
Epoch 61/100
287/287 [==============================] - 48s 164ms/step - loss: 1.0483 -
accuracy: 0.6191 - val_loss: 0.9571 - val_accuracy: 0.6938 - lr: 2.0000e-04
Epoch 62/100
287/287 [==============================] - 48s 165ms/step - loss: 1.0464 -
accuracy: 0.6231 - val_loss: 0.9538 - val_accuracy: 0.6917 - lr: 2.0000e-04
Epoch 63/100
287/287 [==============================] - 48s 163ms/step - loss: 1.0464 -
accuracy: 0.6237 - val_loss: 0.9702 - val_accuracy: 0.6866 - lr: 2.0000e-04
Epoch 64/100
287/287 [==============================] - 48s 164ms/step - loss: 1.0497 -
accuracy: 0.6212 - val_loss: 0.9468 - val_accuracy: 0.6930 - lr: 2.0000e-04
Epoch 65/100
287/287 [==============================] - 49s 165ms/step - loss: 1.0349 -
accuracy: 0.6253 - val_loss: 0.9426 - val_accuracy: 0.6968 - lr: 2.0000e-04
Epoch 66/100
287/287 [==============================] - 49s 166ms/step - loss: 1.0390 -
accuracy: 0.6259 - val_loss: 0.9452 - val_accuracy: 0.6963 - lr: 2.0000e-04
Epoch 67/100
287/287 [==============================] - 48s 165ms/step - loss: 1.0535 -
accuracy: 0.6214 - val_loss: 0.9426 - val_accuracy: 0.6973 - lr: 2.0000e-04
Epoch 68/100
287/287 [==============================] - 48s 164ms/step - loss: 1.0333 -
accuracy: 0.6264 - val_loss: 0.9393 - val_accuracy: 0.6930 - lr: 2.0000e-04
Epoch 69/100
287/287 [==============================] - 48s 165ms/step - loss: 1.0328 -
accuracy: 0.6252 - val_loss: 0.9515 - val_accuracy: 0.6912 - lr: 2.0000e-04
Epoch 70/100
287/287 [==============================] - 48s 165ms/step - loss: 1.0302 -
accuracy: 0.6285 - val_loss: 0.9441 - val_accuracy: 0.6999 - lr: 2.0000e-04
Epoch 71/100
287/287 [==============================] - 48s 164ms/step - loss: 1.0302 -
accuracy: 0.6258 - val_loss: 0.9490 - val_accuracy: 0.6978 - lr: 2.0000e-04
Epoch 72/100
287/287 [==============================] - 48s 164ms/step - loss: 1.0291 -
accuracy: 0.6267 - val_loss: 0.9479 - val_accuracy: 0.6966 - lr: 2.0000e-04
Epoch 73/100
```

```
287/287 [==============================] - 48s 165ms/step - loss: 1.0157 -
accuracy: 0.6345 - val_loss: 0.9335 - val_accuracy: 0.7080 - lr: 2.0000e-04
Epoch 74/100
287/287 [==============================] - 48s 162ms/step - loss: 1.0176 -
accuracy: 0.6318 - val_loss: 0.9390 - val_accuracy: 0.7039 - lr: 2.0000e-04
Epoch 75/100
287/287 [==============================] - 48s 163ms/step - loss: 1.0202 -
accuracy: 0.6320 - val_loss: 0.9428 - val_accuracy: 0.6958 - lr: 2.0000e-04
Epoch 76/100
287/287 [==============================] - 48s 164ms/step - loss: 1.0149 -
accuracy: 0.6343 - val_loss: 0.9503 - val_accuracy: 0.6961 - lr: 2.0000e-04
Epoch 77/100
287/287 [==============================] - 48s 164ms/step - loss: 1.0119 -
accuracy: 0.6372 - val_loss: 0.9392 - val_accuracy: 0.6986 - lr: 2.0000e-04
Epoch 78/100
287/287 [==============================] - 48s 163ms/step - loss: 1.0154 -
accuracy: 0.6288 - val_loss: 0.9310 - val_accuracy: 0.7073 - lr: 2.0000e-04
Epoch 79/100
287/287 [==============================] - 48s 164ms/step - loss: 1.0049 -
accuracy: 0.6408 - val_loss: 0.9376 - val_accuracy: 0.6943 - lr: 2.0000e-04
Epoch 80/100
287/287 [==============================] - 48s 165ms/step - loss: 1.0126 -
accuracy: 0.6350 - val_loss: 0.9364 - val_accuracy: 0.7004 - lr: 2.0000e-04
Epoch 81/100
287/287 [==============================] - 48s 164ms/step - loss: 1.0109 -
accuracy: 0.6327 - val_loss: 0.9264 - val_accuracy: 0.7024 - lr: 2.0000e-04
Epoch 82/100
287/287 [==============================] - 48s 165ms/step - loss: 0.9964 -
accuracy: 0.6388 - val_loss: 0.9188 - val_accuracy: 0.7073 - lr: 2.0000e-04
Epoch 83/100
287/287 [==============================] - 49s 166ms/step - loss: 1.0031 -
accuracy: 0.6370 - val_loss: 0.9636 - val_accuracy: 0.6935 - lr: 2.0000e-04
Epoch 84/100
287/287 [==============================] - 48s 165ms/step - loss: 0.9973 -
accuracy: 0.6423 - val_loss: 0.9265 - val_accuracy: 0.7047 - lr: 2.0000e-04
Epoch 85/100
287/287 [==============================] - 49s 165ms/step - loss: 0.9983 -
accuracy: 0.6425 - val_loss: 0.9246 - val_accuracy: 0.7055 - lr: 2.0000e-04
Epoch 86/100
287/287 [==============================] - 49s 168ms/step - loss: 0.9982 -
accuracy: 0.6422 - val_loss: 0.9145 - val_accuracy: 0.7118 - lr: 2.0000e-04
Epoch 87/100
287/287 [==============================] - 49s 166ms/step - loss: 0.9936 -
accuracy: 0.6402 - val_loss: 0.9219 - val_accuracy: 0.7073 - lr: 2.0000e-04
Epoch 88/100
287/287 [==============================] - 48s 165ms/step - loss: 0.9975 -
accuracy: 0.6439 - val_loss: 0.9357 - val_accuracy: 0.7022 - lr: 2.0000e-04
Epoch 89/100
```

```
287/287 [==============================] - 48s 164ms/step - loss: 0.9923 -
accuracy: 0.6382 - val_loss: 0.9342 - val_accuracy: 0.7039 - lr: 2.0000e-04
Epoch 90/100
287/287 [==============================] - 48s 165ms/step - loss: 0.9853 -
accuracy: 0.6478 - val_loss: 0.9161 - val_accuracy: 0.7121 - lr: 2.0000e-04
Epoch 91/100
286/287 [============================>.] - ETA: 0s - loss: 0.9877 - accuracy:
0.6435
Epoch 91: ReduceLROnPlateau reducing learning rate to 4.0000001899898055e-05.
287/287 [==============================] - 48s 165ms/step - loss: 0.9879 -
accuracy: 0.6434 - val_loss: 0.9423 - val_accuracy: 0.7011 - lr: 2.0000e-04
Epoch 92/100
287/287 [==============================] - 48s 165ms/step - loss: 0.9796 -
accuracy: 0.6409 - val_loss: 0.9032 - val_accuracy: 0.7177 - lr: 4.0000e-05
Epoch 93/100
287/287 [==============================] - 49s 166ms/step - loss: 0.9793 -
accuracy: 0.6418 - val_loss: 0.9140 - val_accuracy: 0.7144 - lr: 4.0000e-05
Epoch 94/100
287/287 [==============================] - 48s 164ms/step - loss: 0.9703 -
accuracy: 0.6499 - val_loss: 0.9083 - val_accuracy: 0.7144 - lr: 4.0000e-05
Epoch 95/100
287/287 [==============================] - 48s 164ms/step - loss: 0.9719 -
accuracy: 0.6461 - val_loss: 0.9072 - val_accuracy: 0.7139 - lr: 4.0000e-05
Epoch 96/100
287/287 [==============================] - 48s 165ms/step - loss: 0.9740 -
accuracy: 0.6437 - val_loss: 0.9065 - val_accuracy: 0.7134 - lr: 4.0000e-05
Epoch 97/100
286/287 [============================>.] - ETA: 0s - loss: 0.9623 - accuracy:
0.6487
Epoch 97: ReduceLROnPlateau reducing learning rate to 8.000000525498762e-06.
287/287 [==============================] - 48s 162ms/step - loss: 0.9624 -
accuracy: 0.6486 - val_loss: 0.9128 - val_accuracy: 0.7126 - lr: 4.0000e-05
Epoch 98/100
287/287 [==============================] - 48s 162ms/step - loss: 0.9664 -
accuracy: 0.6516 - val_loss: 0.9011 - val_accuracy: 0.7162 - lr: 8.0000e-06
Epoch 99/100
287/287 [==============================] - 49s 165ms/step - loss: 0.9686 -
accuracy: 0.6485 - val_loss: 0.9016 - val_accuracy: 0.7162 - lr: 8.0000e-06
Epoch 100/100
287/287 [==============================] - 49s 165ms/step - loss: 0.9639 -
accuracy: 0.6520 - val_loss: 0.8990 - val_accuracy: 0.7177 - lr: 8.0000e-06
```



[ ]:
```
# Evaluate the model
loss, accuracy = CNN_augmented.evaluate(test_ds)
print("Test loss:", loss)
print("Test accuracy:", accuracy)
```

```
62/62 [==============================] - 8s 115ms/step - loss: 0.9610 -
accuracy: 0.7061
Pèrdua final sobre les dades de test: 0.9610257148742676
Exactitud final sobre les dades de test: 0.7060768008232117
```

[ ]:
```
total_params = np.sum([np.prod(w.shape) for w in CNN_augmented.
  ↪trainable_weights])
print("Number of trainable parameters:", total_params)
print(f"Training time: {final_time - start_time}")
```

```
Nombre de paràmetres a entrenar: 5631722
Temps d'entrenament: 4859.546278715134
L'accuracy sobre el conjunt de test obtinguda pel model és 0.7060768008232117
```

The number of trainable parameters is 5.631.722, and the training time is about 80 minutes.

The model obtains a 70% accuracy on the test set, a significant improvement. Our accuracy begins to be in the range of acceptable values.

If we take a look at the learning curves, we can see that we have also solved the overfitting problem. Validation curves are better than training ones, and both measures—validation and training—are able to improve at least until a certain limit.

If we wanted to overpass that limit, we could optimize the architecture even more; the way our data

is fed can also be improved; we could train the same model for more epochs since no EarlyStopping was triggered; or we could optimize the learning rate.

We'll try to improve our approach later on.

But imagine we needed almost perfect results, and we could not afford to perform infinite amounts of hyperparameter tuning for all the possible tuneable parameters, such as the number of layers, number of neurons, learning rate, alpha, etcetera.

What we would do is pick up the work somebody has already done for us. We are not the only ones who have had this problem, so there are already designed and trained models that would take months or years for us to design and test.

This is called transfer learning.

We are not the only ones that have tried to process images to classify them before, so we'll take a model already trained specifically for image multi-classification called InceptionV3.

There are many; a popular one that comes to mind is ResNet50. https://es.mathworks.com/help/deeplearning/ref/resnet50.html

Anyway, we'll choose Inception V3. https://keras.io/api/applications/inceptionv3/

We'll download the already-trained model and apply it to our data.

0.5 5. Red Inception V3 and transfer learning.

Since it's very expensive to design and train from zero ultra-complex architectures, we'll use an already designed and trained model. We'll test how it performs in our data without any exposure to the actual data, and later we'll let the model have a little bit of exposure to the actual data (fine-tuning) and then test how it does.

0.5.1 5.1. Transfer Learning and data augmentation

We'll test the InceptionV3 model on our augmented data used in the last example. We are going to profit from the already-trained internal weights and see how they perform.

InceptionV3 has been trained on the ImageNet dataset, and we'll adapt it to our task. Luckily, it was trained using 299x299 images—what a coincidence—in the same format as our previous model.

We'll substitute the fully connected layer(classifier) to meet our requirements; we'll use the already trained weights of the rest of the network, but that layer will be redesigned, and its weights will be retrained.

Since the number of parameters we are going to train is huge, we will augmentate even more so we can get a better grasp of the general patterns and trends.

Our augmentation model will now consist of:

- Randomflip, will horizontally flip our images.
- RandomRotation with a factor of 0.1.
- RandomZoom with a factor of 0.1.
- RandomContrast with a factor of 0.1.

Let's pass some images through this block and see how they look.

[ ]:
```python
# Augmentation model
augmentation_model = Sequential([
    RandomFlip("horizontal", input_shape=(299,299,3)),
    RandomRotation(0.1),
    RandomZoom(0.1),
    RandomContrast(0.1)
])
```

[ ]:
```python
# Plot some images passed through the model
for images, _ in train_ds.take(1):
    sample_images = images
    break

# Pass images through the model
output_images = augmentation_model(sample_images, training=True)

# Plot the images
plt.figure(figsize=(10, 5))
for i in range(4):
    plt.subplot(2, 4, i + 1)
    plt.imshow(sample_images[i].numpy().astype("uint8"))
    plt.axis('off')
    plt.title('Original image')

    plt.subplot(2, 4, i + 5)
    plt.imshow(output_images[i].numpy().astype("uint8"))
    plt.axis('off')
    plt.title('Preprocessed image')
```

In order to utilize InceptionV3, we must preprocess our data as it was done during its original training. Keras provides us with a function in order to do so:

inception_v3.preprocess_input

It will scale pixel data in a range of -1 to 1. This function must be applied to data in 0 to 255 format.

Our model will look like:

- Augmentation layer.
- Preprocess input layer. Rescale pixels.
- The InceptionV3 feature extractor has the trained weights frozen.
- A GlobalAveragePooling2d layer to abstract features.

This will be our input processing and feature extraction.

The classifier will look like this:

- A fully connected layer consisting of 1024 neurons using the relu activation function.
- A fully connected layer consisting of 10 neurons using the soft max activation function to output the probabilities for each class.

We are going to compile our model with the following settings:

- RMSprop optimizer with a learning rate of 0.00001 (1e-4), specifically used in InceptionV3.
- Train during 100 epochs using EarlyStopping with patience of 10 epochs and monitoring the loss in the validation set; we'll keep the weights of the best model (classifier ones; the others do not change).
- Use ReduceLROnPlateau to monitor the loss function in the validation set with a factor of 0.2 and a learning rate of 0.0000001(1e-6).
- Monitor the accuracy during training and validation.

[ ]:
```
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input

# Load the training model with frozen weights and without the classificator
base_model = InceptionV3(weights='imagenet', include_top=False,␣
 ↪input_shape=(299, 299, 3))
base_model.trainable = False
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicatio
ns/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5
87910968/87910968 [==============================] - 3s 0us/step
```

[ ]:
```
# Define our model
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense, Lambda
model = Sequential([
    augmentation_model,
    Lambda(tf.keras.applications.inception_v3.preprocess_input),
    base_model,
```

```
    GlobalAveragePooling2D(),
    Dense(1024, activation='relu'),
    Dense(10, activation='softmax')
])
```

[ ]:
```
# Compile the model
model.compile(optimizer=RMSprop(learning_rate=1e-4),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
early_stopping = EarlyStopping(monitor='val_loss', patience=10,␣
 ↪restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', patience=5, factor=0.2,␣
 ↪min_lr=1e-6)
```

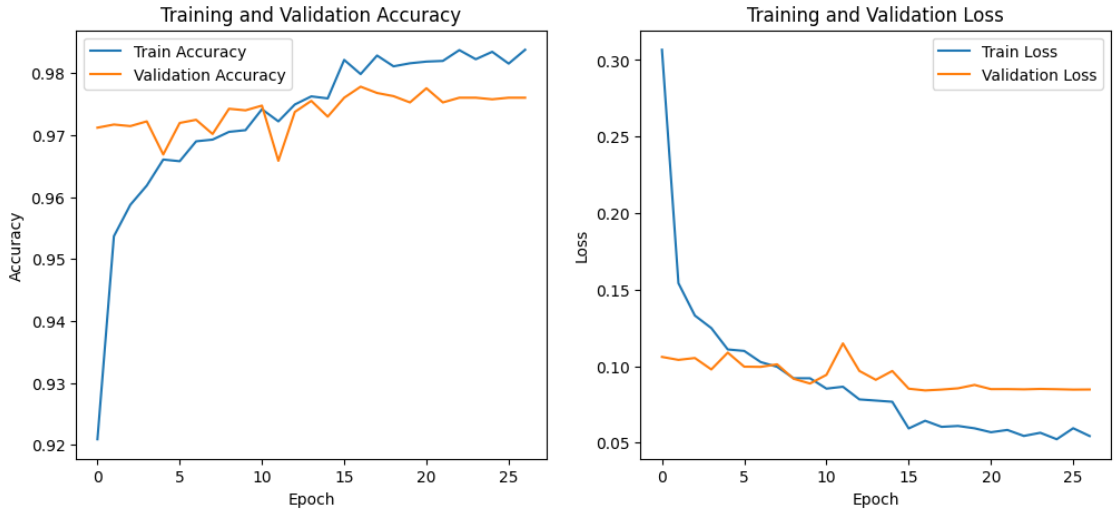
[ ]:
```
# Train the model
start_time = time.time()
history = model.fit(train_ds, epochs=100, validation_data=val_ds,␣
 ↪callbacks=[early_stopping, reduce_lr])
final_time = time.time()

# Plot learning curves
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
```

```
Epoch 1/100
287/287 [==============================] - 63s 188ms/step - loss: 0.3067 -
accuracy: 0.9209 - val_loss: 0.1060 - val_accuracy: 0.9712 - lr: 1.0000e-04
Epoch 2/100
287/287 [==============================] - 51s 173ms/step - loss: 0.1542 -
accuracy: 0.9537 - val_loss: 0.1041 - val_accuracy: 0.9717 - lr: 1.0000e-04
Epoch 3/100
287/287 [==============================] - 50s 171ms/step - loss: 0.1330 -
```

```
accuracy: 0.9588 - val_loss: 0.1053 - val_accuracy: 0.9715 - lr: 1.0000e-04
Epoch 4/100
287/287 [==============================] - 50s 170ms/step - loss: 0.1248 -
accuracy: 0.9619 - val_loss: 0.0978 - val_accuracy: 0.9722 - lr: 1.0000e-04
Epoch 5/100
287/287 [==============================] - 50s 171ms/step - loss: 0.1109 -
accuracy: 0.9661 - val_loss: 0.1088 - val_accuracy: 0.9669 - lr: 1.0000e-04
Epoch 6/100
287/287 [==============================] - 50s 170ms/step - loss: 0.1099 -
accuracy: 0.9658 - val_loss: 0.0996 - val_accuracy: 0.9720 - lr: 1.0000e-04
Epoch 7/100
287/287 [==============================] - 49s 169ms/step - loss: 0.1027 -
accuracy: 0.9690 - val_loss: 0.0995 - val_accuracy: 0.9725 - lr: 1.0000e-04
Epoch 8/100
287/287 [==============================] - 51s 175ms/step - loss: 0.0995 -
accuracy: 0.9693 - val_loss: 0.1012 - val_accuracy: 0.9702 - lr: 1.0000e-04
Epoch 9/100
287/287 [==============================] - 53s 179ms/step - loss: 0.0921 -
accuracy: 0.9705 - val_loss: 0.0917 - val_accuracy: 0.9743 - lr: 1.0000e-04
Epoch 10/100
287/287 [==============================] - 51s 174ms/step - loss: 0.0921 -
accuracy: 0.9708 - val_loss: 0.0886 - val_accuracy: 0.9740 - lr: 1.0000e-04
Epoch 11/100
287/287 [==============================] - 51s 176ms/step - loss: 0.0853 -
accuracy: 0.9742 - val_loss: 0.0943 - val_accuracy: 0.9748 - lr: 1.0000e-04
Epoch 12/100
287/287 [==============================] - 51s 172ms/step - loss: 0.0865 -
accuracy: 0.9722 - val_loss: 0.1148 - val_accuracy: 0.9659 - lr: 1.0000e-04
Epoch 13/100
287/287 [==============================] - 52s 176ms/step - loss: 0.0783 -
accuracy: 0.9750 - val_loss: 0.0969 - val_accuracy: 0.9738 - lr: 1.0000e-04
Epoch 14/100
287/287 [==============================] - 53s 181ms/step - loss: 0.0775 -
accuracy: 0.9763 - val_loss: 0.0911 - val_accuracy: 0.9755 - lr: 1.0000e-04
Epoch 15/100
287/287 [==============================] - 52s 177ms/step - loss: 0.0767 -
accuracy: 0.9759 - val_loss: 0.0968 - val_accuracy: 0.9730 - lr: 1.0000e-04
Epoch 16/100
287/287 [==============================] - 52s 177ms/step - loss: 0.0593 -
accuracy: 0.9822 - val_loss: 0.0853 - val_accuracy: 0.9761 - lr: 2.0000e-05
Epoch 17/100
287/287 [==============================] - 52s 178ms/step - loss: 0.0643 -
accuracy: 0.9799 - val_loss: 0.0841 - val_accuracy: 0.9778 - lr: 2.0000e-05
Epoch 18/100
287/287 [==============================] - 52s 178ms/step - loss: 0.0603 -
accuracy: 0.9829 - val_loss: 0.0847 - val_accuracy: 0.9768 - lr: 2.0000e-05
Epoch 19/100
287/287 [==============================] - 52s 175ms/step - loss: 0.0609 -
```

```
accuracy: 0.9811 - val_loss: 0.0855 - val_accuracy: 0.9763 - lr: 2.0000e-05
Epoch 20/100
287/287 [==============================] - 52s 178ms/step - loss: 0.0594 -
accuracy: 0.9816 - val_loss: 0.0877 - val_accuracy: 0.9753 - lr: 2.0000e-05
Epoch 21/100
287/287 [==============================] - 52s 178ms/step - loss: 0.0568 -
accuracy: 0.9819 - val_loss: 0.0850 - val_accuracy: 0.9776 - lr: 2.0000e-05
Epoch 22/100
287/287 [==============================] - 52s 179ms/step - loss: 0.0583 -
accuracy: 0.9820 - val_loss: 0.0850 - val_accuracy: 0.9753 - lr: 2.0000e-05
Epoch 23/100
287/287 [==============================] - 52s 178ms/step - loss: 0.0544 -
accuracy: 0.9837 - val_loss: 0.0848 - val_accuracy: 0.9761 - lr: 4.0000e-06
Epoch 24/100
287/287 [==============================] - 52s 177ms/step - loss: 0.0565 -
accuracy: 0.9823 - val_loss: 0.0851 - val_accuracy: 0.9761 - lr: 4.0000e-06
Epoch 25/100
287/287 [==============================] - 52s 178ms/step - loss: 0.0522 -
accuracy: 0.9835 - val_loss: 0.0849 - val_accuracy: 0.9758 - lr: 4.0000e-06
Epoch 26/100
287/287 [==============================] - 52s 178ms/step - loss: 0.0594 -
accuracy: 0.9816 - val_loss: 0.0847 - val_accuracy: 0.9761 - lr: 4.0000e-06
Epoch 27/100
287/287 [==============================] - 52s 178ms/step - loss: 0.0543 -
accuracy: 0.9838 - val_loss: 0.0847 - val_accuracy: 0.9761 - lr: 4.0000e-06
```

[ ]:
```
<matplotlib.legend.Legend at 0x790f6b96f9a0>
```



[ ]:
```
# Evaluate the model
loss, accuracy = model.evaluate(test_ds)
```

```
print(f'Test Loss: {loss:.4f}')
print(f'Test Accuracy: {accuracy:.4f}')
```

```
62/62 [==============================] - 10s 153ms/step - loss: 0.0524 -
accuracy: 0.9850
Test Loss: 0.0524
Test Accuracy: 0.9850
```

[ ]:
```
total_params = np.sum([np.prod(w.shape) for w in model.trainable_weights])
print("Number of trainable weights:", total_params)
print(f"Training time: {final_time - start_time}")
```

```
Nombre de paràmetres a entrenar: 2108426
Temps d'entrenament: 1493.575722694397
L'accuracy sobre el conjunt de test obtinguda pel model és 0.9849987030029297
```

Our trainable weights are less than in the previous neural networks, and you could think, if this is a deeper and more complex neural network, why does it have fewer trainable parameters?

Well, we have frozen the weights; we are only training the classifier, so most of the weights of our model are not trainable.

The training time is 25 minutes.

The model gets a 98.5% accuracy, which is astonishing and nearly unimprovable.

Since the weights have been previously trained, we can see that since the first epoch, we are getting excellent results.

The training curves are almost perfect; they follow similar trends and are nearly identical to each other, so there is no overfitting.

The initial model does provide astonishing results, but it's also able to keep learning and improve performance, which is even better.

The training curve takes slightly better values than the validation curve, which is perfectly fine.

0.5.2 5.2. Fine-tuning

We'll use now fine-tuning.

Fine-tuning is basically retraining the whole network using the already-trained weights, exposing them to the training data, optimizing them during a low number of epochs, and using a low learning rate.

So basically, we are giving the feature extractor (where the weights were frozen) the opportunity to learn a bit about the specifics of the actual data.

We'll use the same model.

The compiling settings will vary:

- Adam optimizer with a learning rate of 0.000001(1e-5).

- Training for 10 epochs using Early Stopping with patience of 5 epochs and monitoring the accuracy in the validation set, keeping the weights of the best model.
- ReduceLROnPlateau monitoring the loss function of the validation set and with 3 epochs as patience, factor of 0.2 and a learning rate of 0.00000001(1e-7).
- Monitor the accuracy and loss during training for both datasets.

[ ]:
```
# Define the model
fine_model = InceptionV3(weights='imagenet', include_top=False,␣
 ↪input_shape=(299, 299, 3))
fine_model.trainable = True
global_model = Sequential([
    augmentation_model,
    Lambda(tf.keras.applications.inception_v3.preprocess_input),
    fine_model,
    GlobalAveragePooling2D(),
    Dense(1024, activation='relu'),
    Dense(10, activation='softmax')
])
```

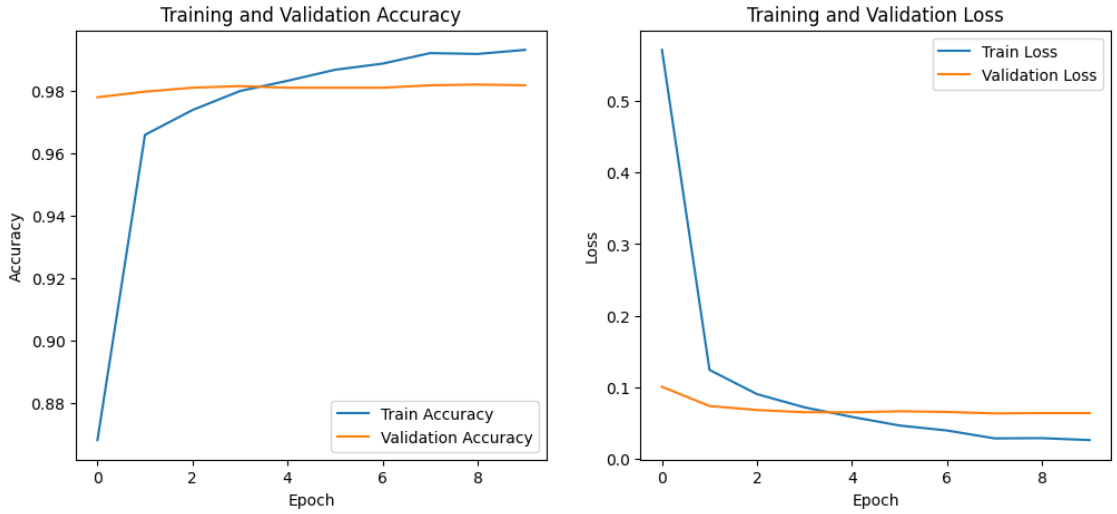
[ ]:
```
# Compile the model
global_model.compile(optimizer=Adam(learning_rate=1e-5),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5,␣
 ↪restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', patience=3, factor=0.2,␣
 ↪min_lr=1e-7)
```

[ ]:
```
# Training
start_time = time.time()
history_fine = global_model.fit(train_ds, epochs=10, validation_data=val_ds,␣
 ↪callbacks=[early_stopping, reduce_lr])
final_time = time.time()

# Plot accuracy and loss curves for both sets.
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history_fine.history['accuracy'], label='Train Accuracy')
plt.plot(history_fine.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history_fine.history['loss'], label='Train Loss')
```

```
plt.plot(history_fine.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
```

```
Epoch 1/10
287/287 [==============================] - 114s 307ms/step - loss: 0.5719 -
accuracy: 0.8680 - val_loss: 0.1004 - val_accuracy: 0.9781 - lr: 1.0000e-05
Epoch 2/10
287/287 [==============================] - 88s 303ms/step - loss: 0.1239 -
accuracy: 0.9660 - val_loss: 0.0734 - val_accuracy: 0.9799 - lr: 1.0000e-05
Epoch 3/10
287/287 [==============================] - 88s 303ms/step - loss: 0.0900 -
accuracy: 0.9740 - val_loss: 0.0678 - val_accuracy: 0.9811 - lr: 1.0000e-05
Epoch 4/10
287/287 [==============================] - 88s 302ms/step - loss: 0.0715 -
accuracy: 0.9801 - val_loss: 0.0647 - val_accuracy: 0.9817 - lr: 1.0000e-05
Epoch 5/10
287/287 [==============================] - 88s 302ms/step - loss: 0.0581 -
accuracy: 0.9834 - val_loss: 0.0646 - val_accuracy: 0.9811 - lr: 1.0000e-05
Epoch 6/10
287/287 [==============================] - 87s 301ms/step - loss: 0.0461 -
accuracy: 0.9869 - val_loss: 0.0661 - val_accuracy: 0.9811 - lr: 1.0000e-05
Epoch 7/10
287/287 [==============================] - 87s 301ms/step - loss: 0.0391 -
accuracy: 0.9889 - val_loss: 0.0651 - val_accuracy: 0.9811 - lr: 1.0000e-05
Epoch 8/10
287/287 [==============================] - 88s 302ms/step - loss: 0.0281 -
accuracy: 0.9923 - val_loss: 0.0631 - val_accuracy: 0.9819 - lr: 2.0000e-06
Epoch 9/10
287/287 [==============================] - 88s 302ms/step - loss: 0.0285 -
accuracy: 0.9920 - val_loss: 0.0635 - val_accuracy: 0.9822 - lr: 2.0000e-06
Epoch 10/10
287/287 [==============================] - 87s 301ms/step - loss: 0.0257 -
accuracy: 0.9933 - val_loss: 0.0635 - val_accuracy: 0.9819 - lr: 2.0000e-06
```

[ ]: <matplotlib.legend.Legend at 0x785a800e6170>



[ ]:
```
# Evaluate the model
loss, accuracy = global_model.evaluate(test_ds)
print(f'Test Loss: {loss:.4f}')
print(f'Test Accuracy: {accuracy:.4f}')
```

```
62/62 [==============================] - 236s 3s/step - loss: 0.0407 - accuracy:
0.9868
Test Loss: 0.0407
Test Accuracy: 0.9868
```

[ ]:
```
total_params = np.sum([np.prod(w.shape) for w in model.trainable_weights])
print("Number of trainable parameters:", total_params)
print(f"Training time: {final_time - start_time}")
```

```
Nombre de paràmetres a entrenar: 2108426
Temps d'entrenament: 902.5978028774261
L'accuracy sobre el conjunt de test obtinguda pel model és 0.9867785573005676
```

The number of trainable weights is 23.876.778. We can see that it has skyrocketed. That is because we have unfrozen the weights of the feature extractor, and now they are trainable.

It took 16 minutes to train for 10 epochs.

We get a 98.68% accuracy, a 0.18% improvement.

This option has been the one providing the best results.

We could conclude that the combination of transfer learning and fine-tuning might be the best option for complex problems when you don't need or are not able to build an ultra-complex neural network from scratch.

Since it leverages an ultra-complex neural network already trained for a similar problem as yours, this means you don't need to design, find the best combination of hyperparameters, and train

the neural network for long days, but you are able to get its performance, and by exposing this neural network a little bit to your actual data, you optimize the results getting top-tier results with minimum effort.

0.6 6. Inference and plotting the results

We are going to analyze the results of our model.

- We are going to pass images through our model, visualize them and check the actual and predicted labels.
- We are going to analyze the typical multi-classification metrics such as precision, recall, f1-score and accuracy.
- We are going to analyze the confusion matrix of our results.

[ ]:
```
# Predictions and labels
# Get from those the confusion matrix
import scikit_learn
pred = []
labe = []

# For each batch of the test dataset
for images, labels in test_ds:
    # Make a prediction
    predictions = global_model.predict(images)
    predicted_classes = np.argmax(predictions, axis=1)
    pred.append(predicted_classes)
    # Get the actual labels
    labe.append(labels)
predictions = [item for sublist in pred for item in sublist]
flattened_labels = [tensor.numpy().flatten().tolist() for tensor in labe]
labels = [item for sublist in flattened_labels for item in sublist]
cm = confusion_matrix(labels, predictions)
```

```
2/2 [==============================] - 0s 55ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 52ms/step
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 53ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 42ms/step
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 44ms/step
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 52ms/step
2/2 [==============================] - 0s 46ms/step
```

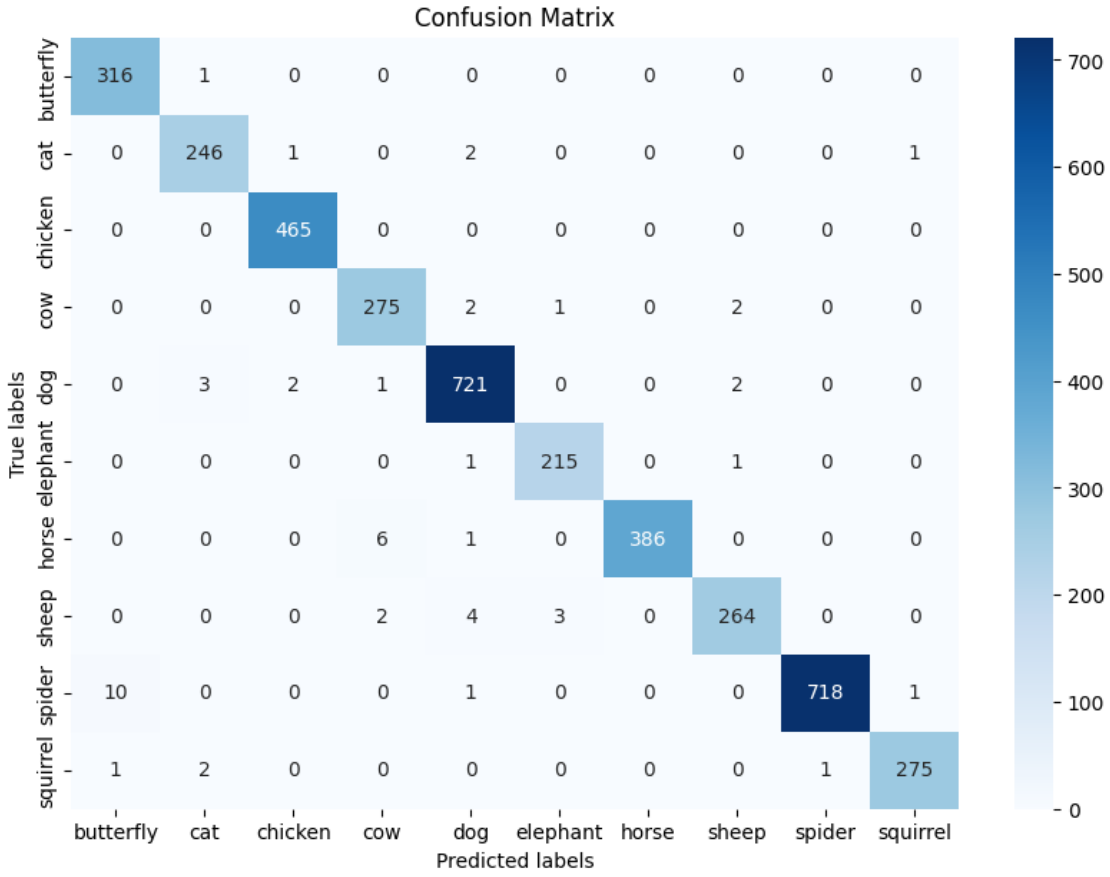
```
2/2 [==============================] - 0s 44ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 42ms/step
2/2 [==============================] - 0s 41ms/step
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 53ms/step
2/2 [==============================] - 0s 42ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 44ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 44ms/step
2/2 [==============================] - 0s 42ms/step
2/2 [==============================] - 0s 48ms/step
2/2 [==============================] - 0s 43ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 41ms/step
2/2 [==============================] - 0s 49ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 47ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 44ms/step
2/2 [==============================] - 0s 46ms/step
2/2 [==============================] - 0s 45ms/step
2/2 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 53ms/step
[[316   1   0   0   0   0   0   0   0   0]
 [  0 246   1   0   2   0   0   0   0   1]
```

```
[  0   0 465   0   0   0   0   0   0   0]
[  0   0   0 275   2   1   0   2   0   0]
[  0   3   2   1 721   0   0   2   0   0]
[  0   0   0   0   1 215   0   1   0   0]
[  0   0   0   6   1   0 386   0   0   0]
[  0   0   0   2   4   3   0 264   0   0]
[ 10   0   0   0   1   0   0   0 718   1]
[  1   2   0   0   0   0   0   0   1 275]]
```

[ ]:
```
# Classification metrics
report = classification_report(labels,predictions, target_names=test_ds.
 ↪class_names)
print(report)
```

```
              precision    recall  f1-score   support

   butterfly       0.97      1.00      0.98       317
         cat       0.98      0.98      0.98       250
     chicken       0.99      1.00      1.00       465
         cow       0.97      0.98      0.98       280
         dog       0.98      0.99      0.99       729
    elephant       0.98      0.99      0.99       217
       horse       1.00      0.98      0.99       393
       sheep       0.98      0.97      0.97       273
      spider       1.00      0.98      0.99       730
    squirrel       0.99      0.99      0.99       279

    accuracy                           0.99      3933
   macro avg       0.98      0.99      0.99      3933
weighted avg       0.99      0.99      0.99      3933
```

[ ]:
```
# Plot the correlation matrix
def plot_confusion_matrix(cm, class_names):
  df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)

  plt.figure(figsize=(10,7))
  sns.heatmap(df_cm, annot=True, cmap="Blues", fmt="d")
  plt.title('Confusion Matrix')
  plt.xlabel('Predicted labels')
  plt.ylabel('True labels')
  plt.show()

plot_confusion_matrix(cm, test_ds.class_names)
```



The main classification metrics are:

- Precision: TP/(TP+FP) Of all the samples predicted as positive, how many of them were real positives?
- Recall: TP/(TP+FN) Of all the labeled positive samples, how many did we predict?
- F1-Score: Is a combination of precision and recall in order to evaluate the model from a global vision. The best possible value is 1, and the worst is 0.

Usually there is a trade-off between precision and recall, specially if the dataset is skewed to a certain class. As you increase one, you decrease the other.

It's not our case, as our sample is not skewed and our model performs near perfection.

We are evaluating a multi-classification problem; this metrics will take a value for each class, where positive means samples that are labeled as the class and negative all the other samples.

All metrics are nearly unimprovable, and the model is superb at its task.

We can observe in the confusion matrix that the great majority of the samples are situated on the diagonal of the matrix, which indicates that they were classified correctly.

Just to comment on something, we could say that butterfly and spider are the classes that are more difficult to discriminate between them, as it's the value where more samples are mislabeled.

This might be due to the unique structure of the photos, which are more similar between each other than to the other classes.

In a nutshell, the results are nearly unimprovable.

We will now get some random images, print the image, make a prediction and compare it with the actual label using our fine-tuned model.

[ ]:
```
# We take 5 random images
for images, labels in test_ds.take(5):
    # Make the prediction
    predictions = global_model.predict(images)
    predicted_classes = np.argmax(predictions, axis=1)

    # Plot the image
    plt.imshow(images[0].numpy().astype("uint8"))
    plt.axis("off")
    plt.show()

    # Print the predicted and actual label
    print("Predicted Class:", predicted_classes[0])
    print("Actual Label:", labels[0].numpy())
```

```
2/2 [==============================] - 0s 53ms/step
```

```
Predicted Class: 6
Actual Label: 6
2/2 [==============================] - 0s 57ms/step
```

```
Predicted Class: 0
Actual Label: 0
2/2 [==============================] - 0s 49ms/step
```

```
Predicted Class: 1
Actual Label: 1
2/2 [==============================] - 0s 56ms/step
```

```
Predicted Class: 3
Actual Label: 3
2/2 [==============================] - 0s 48ms/step
```

```
Predicted Class: 1
Actual Label: 1
```

All predictions were correct; this was expected since our model has nearly a 99% accuracy of 3933 images, and we are picking just 5.

0.7 7. Improve the results of our hand-made model

We are going to make changes in our last proposed architecture and the way our data is fed to the model in order to try to improve the results.

[ ]:
```
# Define training dataset
# Reduce the batch_size to 32, a common size for computer vision problems.
# We will be able to optimize the loss function the double of teams for each
# epoch.

train_dir = '/content/drive/MyDrive/images/train'
image_size = (299, 299)
train_ds = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    validation_split=0.1764,
    subset="training",
    seed=42,
    image_size=image_size,
```

```
    batch_size=32
)

# Validation

val_ds = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    validation_split=0.1764,
    subset="validation",
    seed=42,
    image_size=image_size,
    batch_size=32
)

# Test

test_dir = '/content/drive/MyDrive/images/test'
test_ds = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    seed=42,
    image_size=image_size,
    batch_size=32
)

# Class names

class_names = test_ds.class_names

# We will change the augmentation to the one used for InceptionV3.

augmentation_model = Sequential([
    RandomFlip("horizontal", input_shape=(299,299,3)),
    RandomRotation(0.1),
    RandomZoom(0.1),
    RandomContrast(0.1)
])

# Define the model

my_CNN = Sequential([
    # Feature extractor
    Rescaling(1./255, input_shape=(299, 299, 3)),
        augmentation_model,
    Conv2D(16, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(32, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
```

```
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
        # Add a convolutional layer followed by a maxpooling2d
        Conv2D(128, (3,3), padding='same', activation='relu'),
        MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.2),
    # Classifier
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(len(class_names), activation='softmax')
])


# Compile the model
my_CNN.compile(optimizer=Adam(learning_rate=0.001),
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

# Training
start_time = time.time()
myCNN_history = my_CNN.fit(train_ds,
                    epochs=100,
                    validation_data=val_ds,
                    callbacks=[
                        EarlyStopping(patience=10, monitor='val_loss',␣
 ↪restore_best_weights=True),
                        ReduceLROnPlateau(patience=5, factor=0.2, min_lr=1e-6,␣
 ↪monitor='val_loss', verbose=1)
                    ])
final_time = time.time()
```

```
Found 22253 files belonging to 10 classes.
Using 18328 files for training.
Found 22253 files belonging to 10 classes.
Using 3925 files for validation.
Found 3933 files belonging to 10 classes.
Epoch 1/100
573/573 [==============================] - 1255s 2s/step - loss: 2.0769 -
accuracy: 0.2578 - val_loss: 2.0059 - val_accuracy: 0.2983 - lr: 0.0010
Epoch 2/100
573/573 [==============================] - 40s 70ms/step - loss: 1.7799 -
accuracy: 0.3748 - val_loss: 1.5719 - val_accuracy: 0.4446 - lr: 0.0010
Epoch 3/100
573/573 [==============================] - 39s 67ms/step - loss: 1.5939 -
accuracy: 0.4491 - val_loss: 1.4470 - val_accuracy: 0.5131 - lr: 0.0010
Epoch 4/100
```

```
573/573 [==============================] - 40s 70ms/step - loss: 1.4855 -
accuracy: 0.4892 - val_loss: 1.2984 - val_accuracy: 0.5613 - lr: 0.0010
Epoch 5/100
573/573 [==============================] - 40s 70ms/step - loss: 1.3775 -
accuracy: 0.5275 - val_loss: 1.1697 - val_accuracy: 0.6048 - lr: 0.0010
Epoch 6/100
573/573 [==============================] - 40s 70ms/step - loss: 1.3120 -
accuracy: 0.5558 - val_loss: 1.2068 - val_accuracy: 0.5890 - lr: 0.0010
Epoch 7/100
573/573 [==============================] - 40s 69ms/step - loss: 1.2560 -
accuracy: 0.5746 - val_loss: 1.1220 - val_accuracy: 0.6189 - lr: 0.0010
Epoch 8/100
573/573 [==============================] - 40s 69ms/step - loss: 1.2001 -
accuracy: 0.5916 - val_loss: 1.1413 - val_accuracy: 0.6076 - lr: 0.0010
Epoch 9/100
573/573 [==============================] - 39s 68ms/step - loss: 1.1663 -
accuracy: 0.6050 - val_loss: 1.0776 - val_accuracy: 0.6400 - lr: 0.0010
Epoch 10/100
573/573 [==============================] - 41s 70ms/step - loss: 1.1237 -
accuracy: 0.6242 - val_loss: 1.0973 - val_accuracy: 0.6321 - lr: 0.0010
Epoch 11/100
573/573 [==============================] - 40s 69ms/step - loss: 1.1144 -
accuracy: 0.6293 - val_loss: 1.1550 - val_accuracy: 0.6206 - lr: 0.0010
Epoch 12/100
573/573 [==============================] - 39s 67ms/step - loss: 1.0861 -
accuracy: 0.6351 - val_loss: 1.1362 - val_accuracy: 0.6262 - lr: 0.0010
Epoch 13/100
573/573 [==============================] - 39s 68ms/step - loss: 1.0659 -
accuracy: 0.6404 - val_loss: 1.0785 - val_accuracy: 0.6415 - lr: 0.0010
Epoch 14/100
573/573 [==============================] - 41s 70ms/step - loss: 1.0281 -
accuracy: 0.6551 - val_loss: 1.0098 - val_accuracy: 0.6522 - lr: 0.0010
Epoch 15/100
573/573 [==============================] - 39s 67ms/step - loss: 1.0195 -
accuracy: 0.6540 - val_loss: 0.9514 - val_accuracy: 0.6818 - lr: 0.0010
Epoch 16/100
573/573 [==============================] - 41s 70ms/step - loss: 1.0062 -
accuracy: 0.6637 - val_loss: 1.0247 - val_accuracy: 0.6576 - lr: 0.0010
Epoch 17/100
573/573 [==============================] - 40s 70ms/step - loss: 0.9842 -
accuracy: 0.6716 - val_loss: 1.0508 - val_accuracy: 0.6479 - lr: 0.0010
Epoch 18/100
573/573 [==============================] - 39s 67ms/step - loss: 0.9759 -
accuracy: 0.6733 - val_loss: 0.9107 - val_accuracy: 0.6948 - lr: 0.0010
Epoch 19/100
573/573 [==============================] - 40s 69ms/step - loss: 0.9635 -
accuracy: 0.6789 - val_loss: 0.9606 - val_accuracy: 0.6752 - lr: 0.0010
Epoch 20/100
```

```
573/573 [==============================] - 40s 70ms/step - loss: 0.9500 -
accuracy: 0.6798 - val_loss: 1.0232 - val_accuracy: 0.6634 - lr: 0.0010
Epoch 21/100
573/573 [==============================] - 39s 67ms/step - loss: 0.9405 -
accuracy: 0.6829 - val_loss: 0.9270 - val_accuracy: 0.6961 - lr: 0.0010
Epoch 22/100
573/573 [==============================] - 39s 67ms/step - loss: 0.9236 -
accuracy: 0.6912 - val_loss: 0.8817 - val_accuracy: 0.7037 - lr: 0.0010
Epoch 23/100
573/573 [==============================] - 39s 67ms/step - loss: 0.9113 -
accuracy: 0.6948 - val_loss: 0.9131 - val_accuracy: 0.7075 - lr: 0.0010
Epoch 24/100
573/573 [==============================] - 41s 70ms/step - loss: 0.9090 -
accuracy: 0.6926 - val_loss: 0.9844 - val_accuracy: 0.6713 - lr: 0.0010
Epoch 25/100
573/573 [==============================] - 40s 70ms/step - loss: 0.8863 -
accuracy: 0.7019 - val_loss: 0.8934 - val_accuracy: 0.7073 - lr: 0.0010
Epoch 26/100
573/573 [==============================] - 41s 70ms/step - loss: 0.8821 -
accuracy: 0.7043 - val_loss: 0.8918 - val_accuracy: 0.7075 - lr: 0.0010
Epoch 27/100
573/573 [==============================] - 41s 70ms/step - loss: 0.8874 -
accuracy: 0.7049 - val_loss: 0.8809 - val_accuracy: 0.7057 - lr: 0.0010
Epoch 28/100
573/573 [==============================] - 40s 70ms/step - loss: 0.8740 -
accuracy: 0.7080 - val_loss: 0.9436 - val_accuracy: 0.6938 - lr: 0.0010
Epoch 29/100
573/573 [==============================] - 43s 73ms/step - loss: 0.8732 -
accuracy: 0.7122 - val_loss: 0.9978 - val_accuracy: 0.6668 - lr: 0.0010
Epoch 30/100
573/573 [==============================] - 39s 68ms/step - loss: 0.8585 -
accuracy: 0.7146 - val_loss: 0.8971 - val_accuracy: 0.7024 - lr: 0.0010
Epoch 31/100
573/573 [==============================] - 39s 67ms/step - loss: 0.8570 -
accuracy: 0.7137 - val_loss: 0.8887 - val_accuracy: 0.7113 - lr: 0.0010
Epoch 32/100
573/573 [==============================] - ETA: 0s - loss: 0.8432 - accuracy:
0.7151
Epoch 32: ReduceLROnPlateau reducing learning rate to 0.00020000000949949026.
573/573 [==============================] - 39s 68ms/step - loss: 0.8432 -
accuracy: 0.7151 - val_loss: 0.9371 - val_accuracy: 0.6968 - lr: 0.0010
Epoch 33/100
573/573 [==============================] - 39s 68ms/step - loss: 0.7696 -
accuracy: 0.7396 - val_loss: 0.8216 - val_accuracy: 0.7343 - lr: 2.0000e-04
Epoch 34/100
573/573 [==============================] - 39s 68ms/step - loss: 0.7443 -
accuracy: 0.7456 - val_loss: 0.8316 - val_accuracy: 0.7279 - lr: 2.0000e-04
Epoch 35/100
```

```
573/573 [==============================] - 39s 68ms/step - loss: 0.7314 -
accuracy: 0.7532 - val_loss: 0.7967 - val_accuracy: 0.7383 - lr: 2.0000e-04
Epoch 36/100
573/573 [==============================] - 41s 71ms/step - loss: 0.7199 -
accuracy: 0.7561 - val_loss: 0.8225 - val_accuracy: 0.7302 - lr: 2.0000e-04
Epoch 37/100
573/573 [==============================] - 39s 67ms/step - loss: 0.7212 -
accuracy: 0.7545 - val_loss: 0.8229 - val_accuracy: 0.7315 - lr: 2.0000e-04
Epoch 38/100
573/573 [==============================] - 39s 68ms/step - loss: 0.7242 -
accuracy: 0.7562 - val_loss: 0.7790 - val_accuracy: 0.7442 - lr: 2.0000e-04
Epoch 39/100
573/573 [==============================] - 39s 68ms/step - loss: 0.7097 -
accuracy: 0.7611 - val_loss: 0.8189 - val_accuracy: 0.7363 - lr: 2.0000e-04
Epoch 40/100
573/573 [==============================] - 39s 68ms/step - loss: 0.7150 -
accuracy: 0.7587 - val_loss: 0.8009 - val_accuracy: 0.7343 - lr: 2.0000e-04
Epoch 41/100
573/573 [==============================] - 39s 68ms/step - loss: 0.7013 -
accuracy: 0.7646 - val_loss: 0.8198 - val_accuracy: 0.7330 - lr: 2.0000e-04
Epoch 42/100
573/573 [==============================] - 41s 71ms/step - loss: 0.6993 -
accuracy: 0.7651 - val_loss: 0.8183 - val_accuracy: 0.7361 - lr: 2.0000e-04
Epoch 43/100
572/573 [============================>.] - ETA: 0s - loss: 0.7020 - accuracy:
0.7632
Epoch 43: ReduceLROnPlateau reducing learning rate to 4.0000001899898055e-05.
573/573 [==============================] - 39s 68ms/step - loss: 0.7017 -
accuracy: 0.7633 - val_loss: 0.8110 - val_accuracy: 0.7363 - lr: 2.0000e-04
Epoch 44/100
573/573 [==============================] - 41s 71ms/step - loss: 0.6805 -
accuracy: 0.7688 - val_loss: 0.7652 - val_accuracy: 0.7501 - lr: 4.0000e-05
Epoch 45/100
573/573 [==============================] - 41s 71ms/step - loss: 0.6792 -
accuracy: 0.7706 - val_loss: 0.7731 - val_accuracy: 0.7465 - lr: 4.0000e-05
Epoch 46/100
573/573 [==============================] - 41s 70ms/step - loss: 0.6704 -
accuracy: 0.7689 - val_loss: 0.7666 - val_accuracy: 0.7511 - lr: 4.0000e-05
Epoch 47/100
573/573 [==============================] - 41s 71ms/step - loss: 0.6657 -
accuracy: 0.7716 - val_loss: 0.7613 - val_accuracy: 0.7513 - lr: 4.0000e-05
Epoch 48/100
573/573 [==============================] - 40s 70ms/step - loss: 0.6776 -
accuracy: 0.7696 - val_loss: 0.7602 - val_accuracy: 0.7511 - lr: 4.0000e-05
Epoch 49/100
573/573 [==============================] - 41s 70ms/step - loss: 0.6612 -
accuracy: 0.7756 - val_loss: 0.7685 - val_accuracy: 0.7475 - lr: 4.0000e-05
Epoch 50/100
```

```
573/573 [==============================] - 40s 70ms/step - loss: 0.6736 -
accuracy: 0.7712 - val_loss: 0.7610 - val_accuracy: 0.7526 - lr: 4.0000e-05
Epoch 51/100
573/573 [==============================] - 41s 70ms/step - loss: 0.6625 -
accuracy: 0.7724 - val_loss: 0.7700 - val_accuracy: 0.7498 - lr: 4.0000e-05
Epoch 52/100
573/573 [==============================] - 40s 69ms/step - loss: 0.6690 -
accuracy: 0.7713 - val_loss: 0.7669 - val_accuracy: 0.7511 - lr: 4.0000e-05
Epoch 53/100
573/573 [==============================] - ETA: 0s - loss: 0.6744 - accuracy:
0.7729
Epoch 53: ReduceLROnPlateau reducing learning rate to 8.000000525498762e-06.
573/573 [==============================] - 41s 70ms/step - loss: 0.6744 -
accuracy: 0.7729 - val_loss: 0.7721 - val_accuracy: 0.7490 - lr: 4.0000e-05
Epoch 54/100
573/573 [==============================] - 39s 67ms/step - loss: 0.6725 -
accuracy: 0.7717 - val_loss: 0.7547 - val_accuracy: 0.7536 - lr: 8.0000e-06
Epoch 55/100
573/573 [==============================] - 39s 68ms/step - loss: 0.6688 -
accuracy: 0.7752 - val_loss: 0.7547 - val_accuracy: 0.7541 - lr: 8.0000e-06
Epoch 56/100
573/573 [==============================] - 41s 70ms/step - loss: 0.6631 -
accuracy: 0.7758 - val_loss: 0.7534 - val_accuracy: 0.7549 - lr: 8.0000e-06
Epoch 57/100
573/573 [==============================] - 39s 67ms/step - loss: 0.6650 -
accuracy: 0.7779 - val_loss: 0.7512 - val_accuracy: 0.7546 - lr: 8.0000e-06
Epoch 58/100
573/573 [==============================] - 39s 68ms/step - loss: 0.6578 -
accuracy: 0.7758 - val_loss: 0.7529 - val_accuracy: 0.7562 - lr: 8.0000e-06
Epoch 59/100
573/573 [==============================] - 41s 71ms/step - loss: 0.6558 -
accuracy: 0.7798 - val_loss: 0.7535 - val_accuracy: 0.7569 - lr: 8.0000e-06
Epoch 60/100
573/573 [==============================] - 41s 71ms/step - loss: 0.6646 -
accuracy: 0.7741 - val_loss: 0.7602 - val_accuracy: 0.7544 - lr: 8.0000e-06
Epoch 61/100
573/573 [==============================] - 40s 70ms/step - loss: 0.6569 -
accuracy: 0.7779 - val_loss: 0.7526 - val_accuracy: 0.7557 - lr: 8.0000e-06
Epoch 62/100
572/573 [==============================>.] - ETA: 0s - loss: 0.6595 - accuracy:
0.7770
Epoch 62: ReduceLROnPlateau reducing learning rate to 1.6000001778593287e-06.
573/573 [==============================] - 39s 68ms/step - loss: 0.6595 -
accuracy: 0.7770 - val_loss: 0.7565 - val_accuracy: 0.7552 - lr: 8.0000e-06
Epoch 63/100
573/573 [==============================] - 41s 71ms/step - loss: 0.6602 -
accuracy: 0.7783 - val_loss: 0.7556 - val_accuracy: 0.7552 - lr: 1.6000e-06
Epoch 64/100
```

```
573/573 [==============================] - 40s 70ms/step - loss: 0.6588 -
accuracy: 0.7755 - val_loss: 0.7533 - val_accuracy: 0.7567 - lr: 1.6000e-06
Epoch 65/100
573/573 [==============================] - 40s 70ms/step - loss: 0.6586 -
accuracy: 0.7769 - val_loss: 0.7533 - val_accuracy: 0.7564 - lr: 1.6000e-06
Epoch 66/100
573/573 [==============================] - 39s 67ms/step - loss: 0.6619 -
accuracy: 0.7754 - val_loss: 0.7548 - val_accuracy: 0.7557 - lr: 1.6000e-06
Epoch 67/100
573/573 [==============================] - ETA: 0s - loss: 0.6631 - accuracy:
0.7760
Epoch 67: ReduceLROnPlateau reducing learning rate to 1e-06.
573/573 [==============================] - 39s 67ms/step - loss: 0.6631 -
accuracy: 0.7760 - val_loss: 0.7547 - val_accuracy: 0.7554 - lr: 1.6000e-06
```

[ ]:
```
# Evaluate the model
loss, accuracy = my_CNN.evaluate(test_ds)
print("Loss:", loss)
print("Accuracy:", accuracy)
```

```
123/123 [==============================] - 230s 2s/step - loss: 0.7994 -
accuracy: 0.7490
Pèrdua final sobre les dades de test: 0.7993993163108826
Exactitud final sobre les dades de test: 0.7490465044975281
```

We took as a starting point the model proposed in the 4th point, because trying to improve the performance of IncepctionV3 seems a little bit out of scope.

We reduced the batch size to 32, so backpropagation will be done twice as often as for a batch size of 64 at each epoch, so we'll be measuring loss and optimizing double the times we were till now for each epoch.

Furthermore, we add a convolutional layer with 128 neurons following the 64, hoping it will get better abstractions and output them to the classifier. In order to do so, we will also add a max-pooling2d layer following the convolutional one.

The neural network from the 4th point, had an accuracy of 70.6% against the test dataset; our improved neural network has an accuracy of 74.90%, which is quite a good result; we have improved by 4.3%.

The result seems fine; we could try to improve this result, but I feel like it is enough of an example, and the result is quite good. If we wanted better results, if the problem is not extremely complex, it would be better to leverage transfer learning as we did previously in the 4th point.